University of Camerino

School of Advanced Studies

Doctor of Philosophy in Computer Science and Mathematics - XXXV Cycle

Automating the modeling of blockchain based supply chain tracing systems: the *-chain framework

Supervisor

Prof. Stefano Bistarelli

Dott. Paolo Mori

PhD Candidate

Francesco Faloci

Doctoral Examination Committee

Prof. professor

Prof.professor

Academic Year 2023-2024

ABSTRACT OF THE DISSERTATION

Supply chain management (SCM) systems are a crucial element for the success of modern business processes, especially in a globalized market where traceability and transparency become increasingly relevant aspects. In this thesis, we introduce an innovative SCM system framework, called "*-chain," which exploits distributed ledger technology (DLT) to solve traditional issues related to product traceability and transaction security. Through the use of blockchain, and in particular the Ethereum platform, the system enables not only the efficient design and implementation of supply chains but also the automated creation and deployment of smart contracts. These smart contracts are essential for automating and verifying transactions across the supply chain. In the SCM context, the use of blockchain allows for the tracking of every modification and transaction of a good throughout the supply chain steps, ensuring that information regarding origin, processing, and distribution is always accessible and verified. The framework proposed in this thesis allows users to design and implement supply chain management systems in a visual and intuitive way. The design process begins with defining the various assets involved in the production of goods, which include not only physical products but also various types of operations. Actors within the supply chain, such as suppliers, manufacturers, distributors, and retailers, are also defined. The framework provides a web-integrated graphical user interface to visually map the flows of assets and operations between these actors. Additionally, the framework is tested on two use cases, demonstrating its versatility and effectiveness in real-world scenarios.

LIST OF PUBLICATIONS

- S. Bistarelli, F. Faloci, P. Mori 2023. *-chain: a Framework for Automating the Modeling of Blockchain Based Supply Chain Tracing Systems. In Future Gener. Comput. Syst., Volume 149, pp 679-700, 2023, DOI: 10.1016/j.future.2023.07.012

- S. Bistarelli, F. Faloci, C. Taticchi, M. Miculan, P. Mori. 2023. Modeling Carne PRI supply chain with the *-Chain Platform. Proceedings of the Fifth Distributed Ledger Technology Workshop (DLT 2023), CEUR Workshop Proceedings, 2023, Volume 3460

- S. Bistarelli, F. Faloci, C. Taticchi, P. Mori. 2022. Olive Oil as Case Study for the *-Chain Platform. Proceedings of the 4th Workshop on Distributed Ledger Technology (DLT 2022) co-located with the Italian Conference on Cybersecurity 2022 (ITASEC 2022), CEUR Workshop Proceedings, 2022, volume 3166

- S. Bistarelli, F. Faloci, and P. Mori. *.chain: automatic coding of smart contracts and user interfaces for supply chains. In Third International Conference on Blockchain Computing and Applications (BCCA 2021), Tartu, Estonia, November 15-17, 2021, pages 164–171. IEEE, 2021, DOI: 10.1109/BCCA53669.2021.9656987

- S. Bistarelli, F. Faloci, P. Mori. 2021. Towards a graphical DSL for tracing supplychains on blockchain. 4th International Workshop on Future Perspective of Decentralized APPlications (FDAPP2021). In Euro-Par 2021: Parallel Processing Workshops - Euro-Par 2021 International Workshops, Lisbon, Portugal, August 30-31, 2021, Revised Selected Papers. Lecture Notes in Computer Science, Springer, vol 13098, pp 219–229, Springer, DOI: 10.1007/978-3-031-06156-1_18

- S. Bistarelli, I. Mercanti, F. Faloci, and F. Santini. 2021. Highlighting poor anonymity and security practice in the blockchain of Bitcoin. In

Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC '21). Association for Computing Machinery, New York, NY, USA, 265–272. DOI 10.1145/3412841.3441909.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF ACRONYMS

Acron.	Full Name	Sec
SC	Supply Chain	1.1
SCM system	Supply Chain Management system	1.1
DSL	Domain Specific graphical Language	1.1
BT	Blockchain Technology	1.1
DLT	Distributed Ledger Technology	2.2
scGM	supply chain General Model	3.1
scDI	supply chain Design Interface	3.1
scGR	supply chain Graphical Representation	3.1
scJR	supply chain JSON Representation	3.1
scMT	supply chain Model Translator	3.1
scMIB	supply chain Managing Interfaces Builder	3.1
scAI	supply chain Administration Interface	3.1
scPI	supply chain Participant Interface	3.1
RBAC	Role-Based Access Control	4.1
SBP	Soy Beans Producer	4.1
BPMN	Business Process Model and Notation	9.1
CMMN	Case Management Model and Notation	9.1

CHAPTER 1

INTRODUCTION

Selling high quality -and therefore expensive- goods in an highly competitive market like today, requires producers to provide potential customers with additional value besides the products themselves, e.g., by providing evidences of the quality of such products. As a matter of fact, the presence of (trusted) information guaranteeing the quality of a product, such as information about its production chain, could drive consumers decisions toward such product, even if it is more expensive than similar ones. Increasingly, protocols and regulations are being established at national and international levels to ensure that production chains result in high quality goods as end products. For instance, the PDO label (Protected Designation of Origin) is given only to goods that are produced starting from specific raw materials and following specific and well defined production steps, which are regulated by very strict laws. European countries adopt the denomination of quality for their productions using these laws. These geographical regulations of origin aim to protect and favor the productive system and the economy of the territory. Italy is the country with the highest number of PDO products[1]. For instance, in Tuscany (a region in the central Italy), four DPO labels have been defined for olive oil: *Chianti Classico*, *Terre di Siena*, *Lucca* and *Seggiano*. Each of these labels defines a number of constraints on raw materials and production process. As an example, the Chianti Classico PDO label requires that at least 80% of olives come from plants of the Frantoio, Correggiolo, Moraiolo and Leccino varieties, that such olives are processed within three days from picking, that olive presses are located in the olives production area, and that olives must be transported to such presses the same day they are picked.

Hence, to guarantee that a production process is conducted following

[1] https://www.tmdn.org/giview/

given constraints, it is necessary to put in place mechanisms for properly representing the production process and for tracing and registering how it is carried on, in order to be able to show a summary of such process to the final customer to witness the product quality, or even to compare the registered data with the regulations in law of that specific product to obtain a given quality certification. On the one hand, such tracing mechanisms must be easy to set up and to run in the daily operations, still collecting all the relevant data of the production process. On the other hand, the tracing system must provide a number of security features, such as users authentication, users authorization, tamper resistance and data availability over time, in order to be trusted for the final customers and also for the legal entities who issue the product quality certifications.

1.1 Problem Analysis

A Supply Chain (SC) has been defined in [44] as a network of organizations involved in different processes and activities with the aim of producing value (products or services) at the benefit of the ultimate consumer. Focusing on the product/service, an SC represents a sequence of (even complex) operations which transform the initial assets (e.g., raw materials) into intermediate products/services and then into the final one.

Auditability is a fundamental feature of such a tracing system, assuring that the records tracing the actions that have been performed on the SC assets must be always available and that they cannot be deleted or altered.

Supply Chain Management (SCM) systems are software systems that allow to follow and record the execution of the steps of the process represented by a given SC [28, 18]. A number of SCM systems implementations, both from academy and industry, are currently available[2] [3] [4]. A commonly used approach to have auditable SCM systems is to implement them over blockchain infrastructures.

The use of Blockchain Technology (BT) for SCM systems implementation provides a number of advantages. First of all, blockchains natively provide *auditability*, because all the transactions that are registered on the blockchain will be available forever and without the possibility of being cancelled or altered. Moreover, blockchains also offer trusted support for data processing. Keeping track of supply chain tracking information in blockchain systems makes it easier to monitor supply chain phases, because the data observed for each phase is stored permanently with the certainty of being immutable.

Nowadays, SC tracing notarization is among the most used non-financial

[2] www.swarmnetwork.org/

[3] www.ripe.io

[4] aws.amazon.com/it/industrial/supply-chain-management/

blockchain applications. In the literature and on the market[5] [6] [7] there are many software systems and services implementing Supply Chain Management systems, but they are typically focused on one specific supply chain type. For instance, some current applications propose to implement SCM systems using the blockchain technology [42, 1, 39], but, according to the same researches proposing them, the solutions they provided are focused on the specific field they were designed for, and very difficult to be generalised. Hence, designing and developing a blockchain based SCM system is still a non trivial task, which requires expertise both concerning the supply chain principles and the blockchain technology. To promote the usage of blockchain technology in the supply chain ecosystem, an effort is needed to make the SCM systems design and development easier and available to the domain experts who typically are not distributed ledger technology experts as well.

The creation of an SCM system involves several key steps. The domain expert designs the SC schema, where the objects and the related transformation process are mapped, based on the type of production, and taking into account the involvement of several participants having distinct roles in the process. The domain expert also defines the constraints that must be satisfied for each transformation operation of the SC, required to ensure the quality of the specific final product. Then, a digital platform is chosen to implement the system, which should meet the previously designed schema. The selection of appropriate technologies is critical to the success of an SCM system, and the blockchain is a very suitable candiadate, as shown by this thesis.

[5] https://www.swarmnetwork.org/

[6] https://www.ripe.io

[7] https://aws.amazon.com/it/industrial/supply-chain-management/

1.2 Thesis contribution

In the light of the above considerations, this thesis proposes **-chain*, a framework aimed at simplifying and automatising blockchain based SCM systems design and development processes. The **-chain* framework has been presented in [3, 4, 5, 6, 7], and it helps supply chain domain experts to design their SCM systems and to automatically produce the smart contracts implementing them ensuring auditability via blockchain tracing. The proposed framework consists of: *i)* a *general model* aimed at representing the most common types of SCs; *ii)* a graphical *Domain Specific Language (DSL)* representing such model; and *iii)* a *set of tools* able to implement front-end and back-end of a blockchain based SCM system, starting just from the SC graphical representation.

The general model *(i)* is based on an "asset-centric" representation of the SC. In this sense, the tracing units are the goods involved in the SC, that are referred in our model with the name of "Asset", and the transformations operated by the production process are defined as operations on such assets. The alternative models for representing SCs in SCM system focus on the representation of production phases, analysing the production workflow instead of the assets. Hence, these models adopts a "process-centric" representation of the SC, representing the concept of assets as a property of the process, often failing to consider some asset features such as its geolocation.

**-chain* also provides a graphical DSL *(ii)* allowing to represent SCs through the general model. This DSL follows the "asset-centric" logic focusing on the representation of the life-cycle of assets, i.e., of their transformations during the production process phases. The DSL is composed by five different classes of elements: *Asset, Containers, Operations, Properties, and Roles.* The DSL allows an easy representation of various types of SCs: production, sales, procurement and distribution, but it allows to draw any kind of custom SC the user wants to represent, as we will see in the following of this thesis.

The toolset provided by **-chain (iii)* counts a set of functions that simplify the SCM system creation process. This set consists of: a function for translating the SC expressed through the DLS into a structured JSON representation; a function that translates the JSON representation into solidity code; a function that creates graphical interfaces allowing the management of the SCM system; and a library of javascript functions that connect the blockchain solidity code, once deployed, with the graphical interfaces.

Producing a SCM system exploiting the **-chain* framework is a very simple task. **-chain* provides a graphical SC editor that is used by supply chain domain experts to represent their supply chain processes in terms of assets and operations performed on them. Starting from the graphical represen-

tation, the suite of tools of the framework produces a set of smart contract skeletons implementing the logic of the supply chain to be tracked and monitored. Starting from the graphical representation of the supply chain as well, the suite of tools also creates three web interfaces to interact with the SCM system, to be used, respectively, by the supply chain administrator to manage supply chain participants (e.g., registration and roles), by the supply chain participants to invoke SCM system's smart contracts and record the operations executed on the SC assets, and by the final customers to trace the production of the product they bought. A further contribution of this thesis is the application of the **-chain* framework for producing two SCM systems to trace two real and very relevant use cases from the Italian agri-food sector: the olive oil supply chain following the *Olive Oil PDO* normative, and the "Pezzata Rossa" cow meat supply chain following the *Carne PRI* normative. In both use cases we show the processes of implementing the SC scheme, then the automatic generation of the interactive interfaces and the smart contract. In [3, 4] we presented initial development of the model and the DSL, in order to build the framework *-chain. In these studies we have thoroughly explained the creation choices and asset-centric strategy of the model. In [3] we also proved the translation potential of *-chain, showing how the model can translate a well known study case: the soybean SC. In [5] we presented the current version of the model with a detailed description of components and functionalities, we described the three generated interactive interfaces, and we analysed the generated smart contract via cost evaluation of deployment and usage.

1.3 Thesis outline

This thesis is organized as follows: Chapter 2, provides some background notions needed to understand the design of *-chain and its implementation. Chapter 3 describes the supply chain traceability problem and propose the solution through the *-chain framework. In Chapter 4, we show the model used for the *-chain framework, deeply describing its components. In Chapter 5, we define all the translating procedures, proving how the graphical object is paired with the computational one. In Chapter 6, we evaluate the smart contracts generated by the framework, verifying the basic deployment and execution costs of each individual smart contract In Chapter 7, we show the three interfaces generated by the framework, describing for each of them the main sections and the implemented functions. In Chapter 8, we describe two real use cases represented with our framework, solving two agri-food supply chains: Carne PRI and Olive Oil Product Designation of Origin. Chapter 10 draws the conclusions and presents some possible future work to improve our framework.

CHAPTER 2
BACKGROUND

A supply chain is a network between agents, to produce and/or distribute a specific asset, from a starting point to the final destination [36]. An agent is a entity involved in the supply chain including producers, vendors, warehouses, transportation companies, distributions centers, and retailers. The network includes different activities [..]. A supply chain also highlights the steps it takes to get the product or service from its original state to the final transformed form. These steps include moving and transforming raw materials into finished products, transporting those products, and distributing them.

2.1 Supply Chains

An SC can be defined as a (even complex) process consisting of a sequence of operations which transform a set of initial assets into intermediate ones and then into the final product. An scMS is the software system allowing to trace such sequence of operations executed on the assets of the SC. In particular, the participants to the SC invoke a proper function of the scMS each time they completed the execution of an operation on an asset, following the operation flow defined by the SC. Advanced mechanisms can be applied to certify that a task declared as completed by a participant in the scMS has actually been executed.

Several types of SCs have been defined in the scientific literature [49, 45, 23, 46], depending on the specific production/distribution process they are meant to model: **Production supply chains** are designed to organize the creation and transformation of a product; **Distribution supply chains** are meant to trace products in their paths from the producers to the consumers;

Sales supply chains describe the relationships between distribution nodes in the path chain that a product undergoes in its sales or delivery cycle.

2.1.1 SC classification

Several types of supply-chain have been defined in the scientific literature [49], depending on the specific production/distribution process they are meant to model. In the following we report the more relevant supply-chain types.

Procurement process

analyzes and specifies the goods and materials needed for a production cycle. This process analysis defines for example the number and type of products; type of resources used or necessary; relationship between supply and demand for a specific product; and an overall point of view of the various parts that a processes needs. Procurement is the process of finding and acquiring the assets and services needed by an agent to fulfill its business plan. To make a profit, the cost of procuring goods must be less than the amount obtained selling it, minus whatever costs are associated with processing and selling. Hence, the main reason for a procurement system is to ensure that the agent receives goods, services, or works at the best possible price. This means that the following are parts of the procuerement chain study: assessing the quality of a product; identifying product quantity; marking the localization time; making the product available for the main process. Examples of tasks involved in the procurement analysis are: Developing standards of quality, Creating purchase orders, Buying goods, Inventory management [20].
In many cases, the procurement process is mixed into more abstract chains, taking part in the production chains as an element of the same, although it maintains well-defined properties and structures.

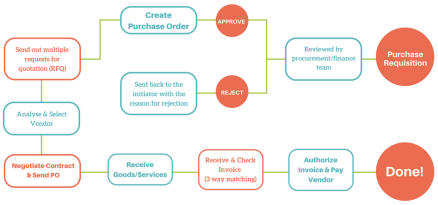


Figure 2.1: Example of procurement chain graph [1]

Procurement processes can be represented by an oriented graph, shown in Figure 2.1, without cycles, which all paths converge on a specific branch of the graph, according to the asset.

Production supply chain

type is designed to organize the creation of a product. This type traces the phases in which the asset under analysis undergoes transformations: from the origin point to the end of the life cycle of production. In some cases, this type of chain describes in detail the changes, the time required for transformations, the information required for production.[49]

Production chain is an analytical tool used to detail the structure of the production process and its transformations. These models generally include production of both goods and services: production process is a sequence of productive activities leading to an end; it uses a chain of linked functions or operators on an asset, each step adds some value to the sequence. Production chain is often called "value-added chain" or "value chains" [29]. The states in the chain are connected through a set of functions: transactions. The organizational or logistical structure of the transactions defines the asset's production. The production chain is often (not correctly) called "production network". It is possible to distinguish these terms: the production supply chain one involves different activities within the transformation or development systems; the production network instead establishes the relationships between the assets in the supply chain.[53]

Figure 2.2: Example a production supply chain scheme [2]

Production supply chain is represented by a flowchart, with bifurcations and cycles, where there is always a well-known (defined) initial state, and possibly one or multiple final states. Figure 2.2 is represented a rough outline of a production flow of an asset. The main components involved are: agents, resources and operations. The purpose of the scheme is to analyze the production chain of a specific object: this object is usually never mentioned, but it is detailed into its components.

[2]Image from `https://anno1800.fandom.com/wiki/Production_chains`

Distribution supply chain

type aims to organize and manage the traceability of resources. A supply chain of this type divides into channels for each macro termination area. This type of supply chain specifies all the agents or the intermediaries through which an asset passes by until they reach the final consumer [49]. Distribution channels can include wholesalers, retailers, deliverers, and even the Internet. A distribution channel is technically a path by which all assets must travel to arrive at the consumer. This chain also describes the payments' way make from the end consumer to the original vendor. Distribution channels depend on the number of intermediaries required to deliver a product to the end point: these assets may make their way to consumers through multiple channels[23][46]. Channels are broken into two different forms: direct and indirect. A direct channel allows the consumer to make purchases from the manufacturer while an indirect channel allows the consumer to buy the goods from a wholesaler or retailer. Indirect channels are typical for goods that are sold in traditional brick-and-mortar stores. With indirect distribution, a product that goes from the manufacturer to a distributor generally takes a price increase during the process.

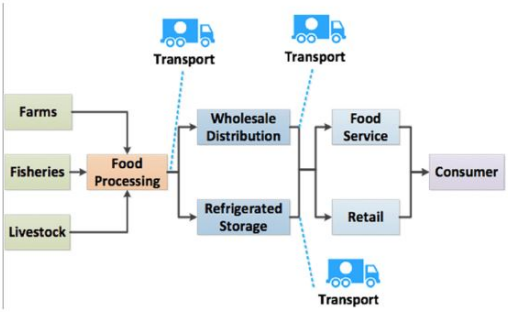


Figure 2.3: Example of a distribution supply chain scheme [47]

Distribution supply chain model is often represented with a network of intermediate nodes, where each node represents a gather point, each path created within the network identifies an instance which is properly designed for the asset under exam; this typology may contain cycles, especially supply chains geared towards redistributing goods or loop-recycling chain [23].

Sales supply chain

type describes the relationships between distribution nodes of an asset; it does not deal with changes in the asset or possible substantial changes, but only with the path chain that a product undergoes in its sales or delivery cycle: generally we speak of a finished product from producer to consumers[49].

This models is a supply chain system which defines the relation between its customers. This kind of chain may represents a sort system that surveys the "agreement" regarding expected offers and demands. Also it may represents a pure market relation with many agents involved. This topology generally tracks back all the trades of many products in order to analyze or specify the asset's life cycle. The typical length of an asset's life cycle significantly influences appropriate marketing, production planning and financial strategies [57][10]. Sales supply chain is represented by a single chain that unites several intermediaries. Each passage between intermediaries establishes a connection between two points in the chain.

2.1.2 Supply Chain Example: the Soy Supply Chain

Figure 2.4 shows an example of a use case of supply chain representing the soybeans life cycle [39], from the seed production phase, to the end customer.

The schema highlights different phases, each of them describing a different type of supply chain: transitions from a point to another characterize moving operation (in green); the passing from an owner to another shows sales phases (in red); the various phases where the object under examination changes its properties are transformation phases (in yellow).

2.2 Blockchain and Smart Contracts

Distributed Ledger Technology (DLT) defines a protocol acting in a decentralized fashion to provide access, validation and updating of a ledger maintained by multiple entities, without assuming trust among them, and the Blockchain technology is one implementation of DLT. More in detail, a Blockchain is a sequence of blocks connected through cryptographycally protected links and embedding a set of transactions. Blockchain users submit the transaction they want to be included in the next blocks (payments for instance), while blockchain miners validate such transactions through a consensus algorithm and insert them in a block. Blockchain main features are decentralization, immutability, and transparency. First generation blockchains were focused on cryptocurrencies (such as Bitcoin[3]). Subsequently, a new generation of blockchains, able to execute programs called smart contracts, were designed. The most famous examples are Ethereum[4] and EOS.IO[5]. The advent of smart contracts paved the way for blockchain based applications in

[2] https://courses.lumenlearning.com/marketing-spring2016/chapter/putting-it-together-place-distribution-channels/

[3] bitcoin.org

[4] ethereum.org/

[5] eos.io/

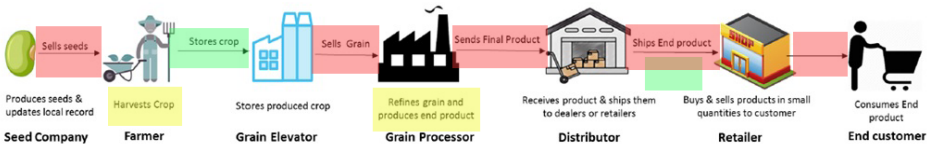


Figure 2.4: Scheme of the Supply chain used on soybeans traceability study [39].

industry and in public sectors [29, 42, 31]. Smart contracts are indeed "self"-executing contracts (scripts) that represent the agreement, written into lines of code, between actors in the blockchain ecosystem (usually a buyer and a seller) .

Smart contracts execution (and the produced transactions) are easily traceable in the blockchain ledger and no trusted third party is needed to prove such agreement execution. Hence, smart contracts permit trusted transactions and agreements among those who are sharing the distributed code. Solidity is a coding languages for smart contracts[6] which is widely used on the Ethereum blockchain, and we use it in the following of this paper as our reference language.

[6] `docs.soliditylang.org/en/v0.8.4/`

CHAPTER 3

*-CHAIN APPROACH

The *-chain framework aims to help its users in designing SCM systems integrated with blockchain technology. In particular, the framework allows to easy represent the SC of interest via a graphical environment and then, through automated functions, it generates a complete SCM system, consisting of a set of smart contracts, javascript modules with interface functions, and graphical interfaces for administrator, participants to the production process, and customers.

The underlying idea of the *-chain approach is to enable the producers, who are the experts of their specific supply chain processes (e.g., the soybeans production process described in Section 2.1.1, or the olive oil production process described in Section 8.1) to contribute with their expertise to define the related blockchain based SCM system, although they do not have (or they have a little) knowledge of the blockchain technology. For this reason, the proposed framework decouples the supply chain process design phase from the smart contract design and programming phase, being typically distinct the two actors in charge of executing such phases. As a matter of fact, the supply chain process design phase will be carried out by an expert of the specific production process, while the related smart contracts programming will be automatically executed by the *-chain framework and, where required, refined by software developers expert in blockchain technology. From the practical point of view, the *-chain framework provides to the experts of the specific supply chain processes an user friendly and, at the same time, powerful, graphical interface for SCM system design and development that allows them to focus on the representation of the workflow of operations of the production process, hiding all the technical and blockchain related details.

3.1 *-chain Framework workflow

The *-chain framework consists of the following components:

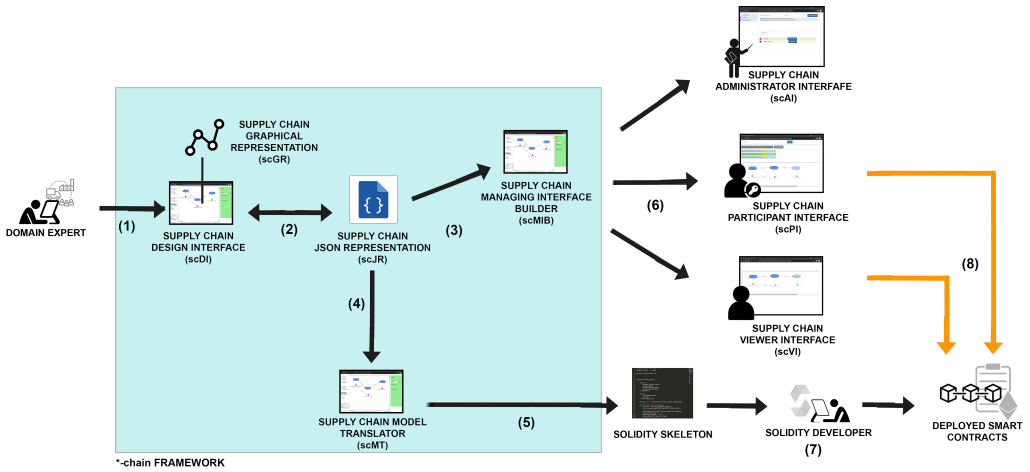


Figure 3.1: General *-chain Framework usage workflow.

- The supply chain General Model (scGM): a general model and the related Domain Specific Graphical Language (DSGL) allowing to represent the most common types of supply chains. We have chosen the graphical representation for the DSGL so as to ease the domain experts who have no computer science competences. As a matter of fact, SC domain experts do not necessarily need to know data structures or programming languages to draw their SC, because the framework has the task to translate the SC designed according to scGM into the data structures and the code implementing the SCM system, and integrating it into a blockchain environment. The SC domain experts basically need to properly chose and combine the constructs or the scGM, by connecting them with one-way arrows, to represent the SC workflow, its properties, and its constraints.

- The supply chain Design Interface (scDI): a graphical editor allowing the SC domain experts to design the SCM systems representing their specific supply chains using the aforementioned DSGL. This editor produces a graphical and a textual JSON-formatted representation of the specific supply chain, which we call, respectively, supply chain Graphical Representation (scGR) and supply chain JSON Representation (scJR). The scJR, being a JSON string, can be saved onto a file and re-used subsequently (e.g., to update the supply chain or to create a new supply chain starting from the current one).

- The supply chain Model Translator (scMT): a tool which, starting from the scJR created with the design interface, derives the software architecture of the smart contracts required for the development of the related SCM system. These smart contracts represent all the asset types that are present in the SC, and each smart contract has a function for tracing the execution of each operation supported by the asset it represents. At this stage of framework development, each operation is considered unitary/atomic and not representative of a set of more complex operations. The framework thus devised is sufficiently scalable to guarantee future developments in this regard.

- The supply chain Managing Interfaces Builder (scMIB): a tool which, starting from the scJR and the related smart contracts skeletons (created from the scMT), creates three graphical user interfaces for the management of the blockchain based SCM system:

  - one graphical interface for SC administrators, the supply chain Administrator Interface (scAI), allowing them to enable other entities to participate to their SCM systems, setting proper roles and rights for them. The scAI is described in detail in Section 7.1.1;
  - one graphical interface for the supply chain participants, the supply chain Participant Interface (scPI), allowing them to register on the SCM system the actions that they have executed on the assets in the SC, according to the roles they have been assigned. The scPI is described in detail in Section 7.1.2.
  - one graphical interface, the supply chain Viewer Interface (scVI), for the customers who bought the final goods, allowing them to check information and operation executed for producing such goods through the SCM system. The scVI is described in detail in Section 7.1.3.

Figure 3.1 shows the architecture of the *-chain framework and the interactions among its components during the SCM system design phase, during the SCM system creation phase, and when it is in usage. In particular, the figure shows that the SC domain expert uses the scDI (step 1) to design its supply chain process using the DSGL. Using our model, the user builds their scGR, enriching the design with constraints and properties related to the production process and assets under consideration. The scGR resulting from the design step is translated in the scJR (step 2), which encapsulates in textual format all the components described in the drawing. The domain expert uses the scMIB (step 3) to automatically produce from the scJR the

graphical user interfaces to be made available to supply chain Administrator, to supply chain participants, and to final customers to interact with the SCM system according to their roles (step 6). In particular, these three interfaces communicate directly with the SCM system smart contracts deployed on the blockchain (steps 8) through specific functions. Moreover, the scJR is also used by the scMT (step 4) to produce the skeleton of the smart contracts that implement the SCM system (step 5). A complex module of the scMT translates each section of the scJR (assets, operations, roles, properties and constraints) producing the smart contracts data structured and methods implementing the SCM system. This skeleton will be used by the Solidity Developers (step 7), who will add the code for managing specific requirements thus obtaining the final smart contracts that will be deployed on the blockchain.

Each step and component of the aforementioned workflow will be described in detail in the following sections.

3.2 Architecture of *-chain SCM systems

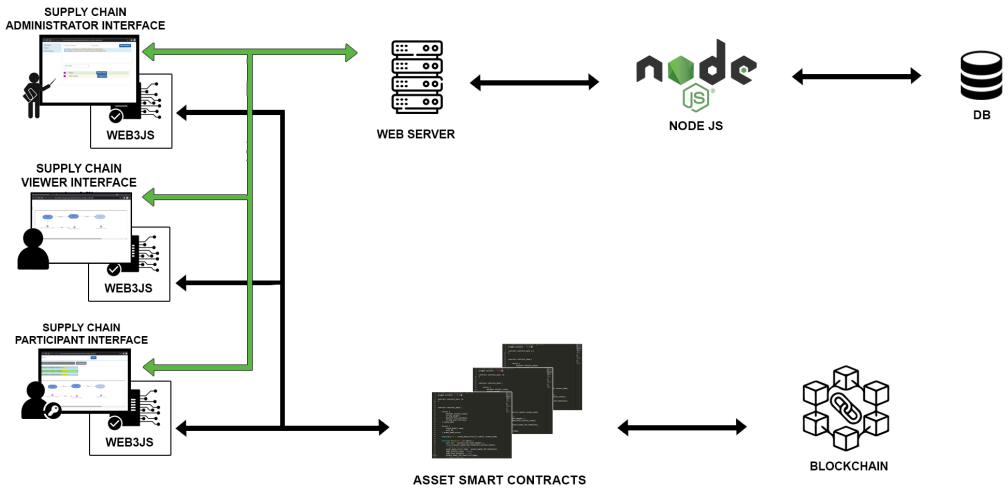


Figure 3.2: application deployment workflow.

Figure 3.2 shows the final architecture of a SCM system produced by the *-chain framework. This architecture is independent from the specific supply chain process that has been designed through the *-chain framework. Users interacts with the SCM system via the three graphical user interfaces, depending on the role they have in the SC scenario. Another component of the SCM system are the smart contracts representing all the assets involved in the SC and their operations. These smart contract are deployed on the

reference blockchain, which -for the sake of example- is the Ethereum virtual machine (EVM) in our case. The last architectural component is a web service, a provider -which for mere convenience we have indicated a service configured with Node.Js- and a local Structured Query Language (SQL) type data base (DB or DBMS), that are used for off-chain data storage.

The three graphical user interfaces loads the same web3js javascript module, which contains a collection of functions used to interface the user with the smart contracts in the blockchain. The data concerning the assets are stored on the blockchain, into the states of the asset themselves, and they can be retrieved exploiting the functions implemented by the related smart contracts, which, in turn, are paired with the functions of the *web3js* module to allow their invocation from the graphical user interfaces. Each graphical user interface resides in a web server that provides it as web service. Each interface needs to connect to a server-side service via a specific provider: to facilitate the framework we assumed a web service built via Node.js. The personal data of users involved in the SCM system are not saved directly on the blockchain. Their storage is delegated to a local DBMS which, for instance, could be deployed on the SC administrator premises. When the graphical user interface needs such sensitive information, the web server queries the local DBMS to retrieve them.

An example of usage of the SCM system is when an SC domain expert with the role of administrator wants to associate a role to a supply chain participant. In particular, through the scAI, the SC domain expert selects a user and a role and invokes the association function. This function invokes a smart contract which, in turn, stores such association in the blockchain.

Another example of usage of the SCM system concerns the user access to the scVI. In particular, the scVI, to show the current status of each asset, loads data from different sources. At first, through a view function (*view()*), it invokes the smart contract related to the specific asset to be viewed, in order to get the current value of the properties and all the operations it has undergone. Subsequently, the scVI get from the web service the sensitive data of the actors involved, that are not stored on the blockchain.

CHAPTER 4

SUPPLY CHAIN MODEL

The definition of a general model for the representation of the objects and of the typical operations involved in supply chains is the first step towards the design of a framework for automatizing the development of blockchain based SCM systems. This model was developed in order to be able to build schemes that can be adapted to supply chain design, having assets (not operations or production steps) as the focus: in this sense it is referred to as asset centric model (as opposed to process centric).

4.1 A General Model and a Domain Specific Graphical Language for Representing Supply Chains

In our model, we identify three families of elements that will be the bricks of any SC:

- *Assets;*
- *Containers;*
- *Operations.*

In addition, we need to consider also *Roles & Rights* to enforce authorizations on the SC operations. Starting from the general model, we define the related DSGL by giving a graphical representation of the aforementioned elements. The DSGL is then used by the *-chain framework users, i.e., the supply chain domains experts, to represent their specific supply chains. For instance, at the end of this chapter we show how the soybeans supply chain

reference example described in Section 2.1.2 can be represented using our DSGL. In the following of this section, we give a detailed description of the elements composing the general model and of their corresponding graphical representations.

4.1.1 Asset

Assets are the objects involved in the process described by the supply chain, on which the operations are executed. For instance, the *Seed* and the *Crop* are two examples of assets involved in our reference example. An asset is *uncountable* if it must be placed into a container in order to be tracked (e.g., olives need to be stored into crates; the olive oil needs to be stored in tins or bottles; soybean seeds need to be contained into sacks).

On the opposite, an asset is *countable* when can be tracked without the need of containers and when it can easily be identified (a car, a cow, a diamond, etc.). An asset is *consumable* if it is destroyed as a consequence of its use or transformation, and *non-consumable* if it can be used more than one time (so an apple tree producing apples is non-consumable). In our reference example, the *Seeds* are uncountable and consumable assets because when they turn to *Plantation* (after some time), they don't exist anymore. A wheel of cheese, a cattle, a pork are all good example of countable and consumable assets; meanwhile a fruit tree, a milk cattle cow, are countable and non-consumable assets. Figure 4.1 shows the graphical representation of asset types where, for instance, a countable and non-consumable asset is represented by a rectangular box with thick borders.

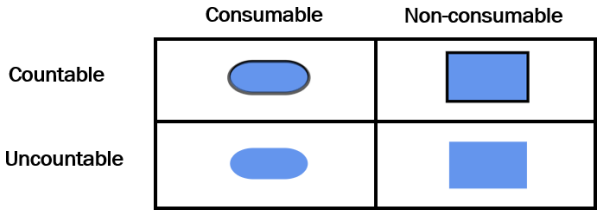


Figure 4.1: Graphical representation of Assets.

Each asset has a number of properties which characterize it. Assets' properties are very important in our approach, because they are used to represent the changes of the state of the assets during the production process. By explicitly declaring which properties change after each operation, the state of the asset can be properly updated after the operation execution. As a matter of fact, the supply chain participant who executes an operation -including the create one- provides the new values of the related properties, in such a way that such properties are updated.

Owner and *controller* are two properties that are always present and that are common to all assets. The owner is the supply chain participant who has the ownership of the asset (e.g., the owner created the asset or they bought that asset from another supply chain participant). The controller, instead, is the supply chain participant who has currently the physical availability of the asset, and hence could execute operations on it. The owner and the controller of an asset could be the same subject in some phases of the supply chain process, and could be distinct subjects in other phases of the supply chain. For instance, after the create operation, the owner and controller of the newly created asset is the same, i.e., the subject who created the asset. Instead, the owner and the controller do not coincide when the owner gives the asset to a third party in charge of performing some kind of transformation on it. The *position* of the asset, i.e., its geographical location, is another property that is always present in each asset.

Each asset has also other properties that are specific to that asset and are represented in the SCM system because they are relevant to describe the asset and the transformations performed in the production process. In our reference example, the asset *Seed* has one relevant property called *weight*. The asset *Olive*, instead, has several properties: PDO, producer ID, date, quality, lot ID, and Cultivar.

In our DSGL, properties are represented by small circles connected to the asset by an edge (as shown in Figure 4.2 and Figure 4.7).

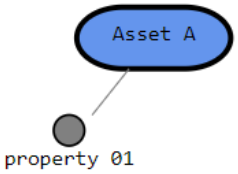


Figure 4.2: A basic example of asset with its property.

4.1.2 Containers

Containers are the components of our model representing what can contain -usually uncountable- assets. Containers are always countable and traceable. For this reason, uncountable assets must be placed into containers in order to be transported and tracked. Countable objects can be placed into containers as well, for instance, to simplify their storage and transportation. Some container examples could be: sacks, boxes, silos, haulers, and ships. In our reference example, *Sack* containers are used for the storage of the *Seed* uncountable asset. Similarly to assets, containers can be consumable or non-consumable. In our DSGL, the representation of containers is different

from assets' representations. Non-consumable containers appear as wider transparent rectangles (the border is the container), while consumable ones have rounded corners. The graphical representation of containers is shown in Figure 4.3.

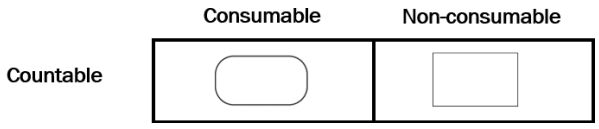


Figure 4.3: Graphical representation of Containers.

Figure 4.4, instead, shows a simple example of consumable container (called "*Package A*") with a property (called *property 01*).

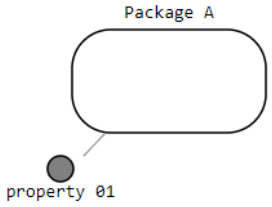


Figure 4.4: A basic example of container with its property.

Finally, to represent that an asset is placed into a container, the box representing the asset is drawn inside the box representing the container. For instance, see Figure Figure 4.7 where the Olive uncountable asset is placed in a Crate container.

4.1.3 Operations

Operations are used to represent modifications or transformations of assets. First of all, for the sake of clarity we recall that, since our framework is meant at defining a tracing system for supply chains, the effect of invoking such operations in our system by a supply chain participant is the one of registering on the blockchain that such participant declares to have executed the corresponding physical operation on the real asset. The handover of a good, the processing of a raw material, the transportation of a package, are all good examples of operations that can be represented by this model. An operation executed on an asset (except for the sell one) can be registered on the SCM system only by the supply chain participant having the physical availability of the asset, i.e., the controller, provided that such controller holds the right to perform such operation (as described in the following of this section). When designing a supply chain, the supply chain domain expert defines the operations representing such supply chain choosing among the operation types

defined by our model, and giving to each specific operation its own mnemonic name (in the smart contract code, the operation name has the asset name as prefix, while in the scGR the prefix is not reported). For instance, in our reference example, the transformation operation which transforms the *Plantation* asset in *Crop* asset is called *Plantation_Harvesting.* Instead, since there is only one *sell* and one *giveControl* operation for each asset, and since they have the same meaning for all the assets, such operations maintain their original names independently on the asset they are applied to.

Our DSGL represents each operation type with a distinct graphical format, as shown in the following figures, and the scJR representing a specific supply chain reports the features of each of the operation that have been defined, including the operation type. In the following, we describe the main types of operations defined by the proposed model:

- *asset_create* (Figure 4.5) and *asset_destroy* (Figure 4.6): these operations are applied, respectively, to create a new asset and to destroy an existing one. The supply chain participant who invokes the *asset_create* operation will be the owner and the controller of such asset, and they must provide the values of the properties declared in this operation for the new asset. Figure 4.7 shows an example of *asset_create* operation for the *Olive* asset, called *Olive_HarvestinLot.* In case of execution of a destroy operation, no further operations can be executed on that asset.

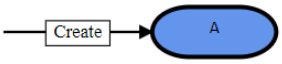


Figure 4.5: A basic example of asset_create() operation.

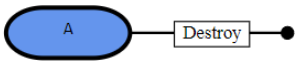


Figure 4.6: A basic example of asset_destroy() operation.

- *asset_update* (Figure 4.8): this operation concerns the update of some properties of the asset, i.e., this operation modifies the properties of the asset, such as its weight, its color, its height. This operation does not change the asset nature or its original traits. In our reference scenarios, we defined several *asset_update* operations to represent asset's properties changes. For instance, the "*Conservation*" operation, labeled (2) in Figure 8.6, places the asset olive in the crate container in a proper conservation area, updating the localization data and the temperature of the olives.

- *asset_move* (Figure 4.9): this operation concerns the update of the position property. Thus, this operation modifies the geographical location

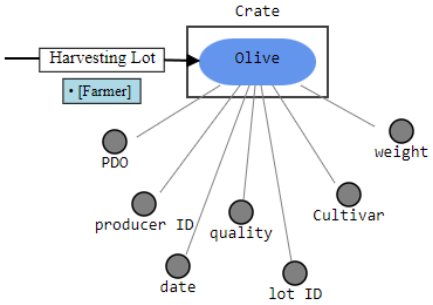


Figure 4.7: An example of asset_create() operation, refering to Figure 8.6 use case.

Figure 4.8: A basic example of asset_update() operation.

property. This operation is a specific instance of the update operation ones.

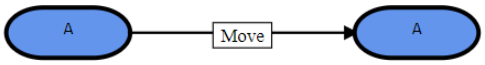


Figure 4.9: A basic example of asset_move() operation.

- *asset_pack* (Figure 4.10): this operation represents the packaging of an asset inside a container, suitable for transportation and tracking. All kinds of assets (countable/uncountable and consumable/non-consumable) or objects can be stored in containers (for the uncountable assets it is mandatory, while for countable asset it is discretionary).

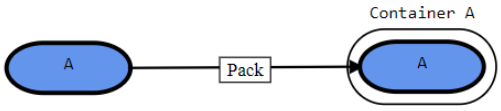


Figure 4.10: A basic example of asset_pack() operation.

This operation could be repeated several times, storing different container object inside another. Figure 4.11 describes an asset_pack() operation, called *load* on the *Cow* asset; the asset is loaded into a container with the name *Truck*.

- *asset_unpack* (Figure 4.12): this operation represents the unpacking of assets from a container. This operation pulls out the content from a previous

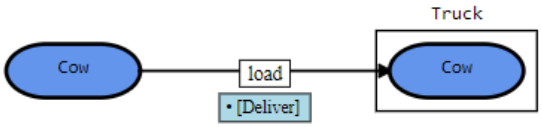


Figure 4.11: An example of asset_pack() operation of the Carne PRI supply chain, described in Section 8.2.

loaded container. Consumable containers are destroyed by the unpacking operation, while non-consumable ones become empty, and could be reused for packing other assets.

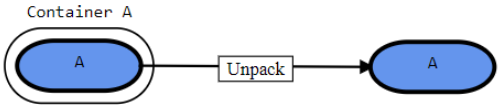


Figure 4.12: A basic example of asset_unpack() operation.

- *asset_flow* (Figure 4.13): this operation is used to move an asset from a container into another. In this case, the asset does not undergo any modification or transformation, the containers where the asset is stored are what is changed by this operation. An example of the asset_flow operation is the "Bottling" operation, shown in Figure Figure 4.14 (taken from Figure 8.9). The "Bottling" operation transfers the asset Oil from a "Containing Tank" (countable, non-consumable) to "Bottles" containers (countable, consumable).

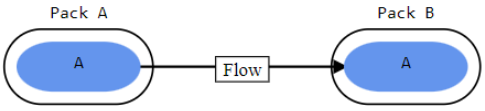


Figure 4.13: A basic example of asset_flow() operation.

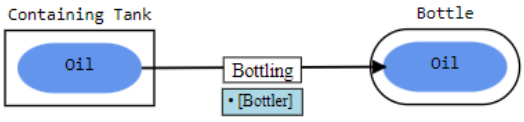


Figure 4.14: An example of asset_flow() operation of the PDO Olive Oil supply chain, described in Section 8.1.

- *asset_transform* (Figure 4.15): the transformation operations are meant to represent the execution of operations which, starting from and asset,

creates another kind of asset. Specific transformation operations are defined for each asset in each supply chain. As an example, in the soybeans supply chain we defined the *Plantation_Harvesting* transformation operation, which transforms *Plantation* assets into *Crop* assets. Transformation operations are different for consumable and non-consumable assets. In particular, when applied to a consumable asset, the operation represents the destruction of the original asset and the creation of a new one (or ones). Instead, when applied to a non-consumable asset, the operation represents the generation of a new asset (or assets), i.e., the original asset still exists, and a new asset is created. In both the previous cases, the *asset_transform* operation is meant to represent the execution of transformations which are not reversible in that supply chain. This means that, in case of consumable assets, it is not possible to have back the original asset from the newly created one. In case of non-consumable asset, instead, it is not possible to recompose the asset on which the operation has been executed and the newly created one obtaining the original asset (however, is possible to destroy the newly created asset). The owner and the controller of the newly created asset are set to be the same as the originating asset.

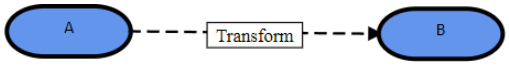


Figure 4.15: A basic example of asset_transform() operation.

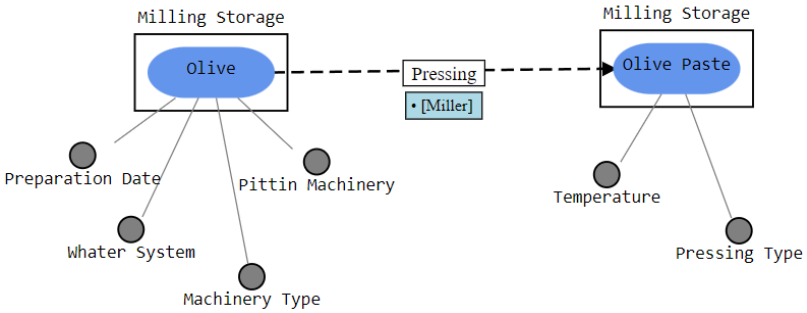


Figure 4.16: An example of asset_transform() operation related to the PDO Olive Oil supply chain, described in Section 8.1.

Figure 4.16 describes an example of transform operation related to the PDO Olive Oil supply chain, described in Section 8.1. The *Olives* contained in the *Milling Storage* are pressed (using the *Pressing* operation) in order to to generate the *Olive Paste*.

- *asset_monitor* (Figure 4.17): the monitor operation is aimed at recording on the scMS some relevant information concerning an asset. For instance, a supply chain participant could measure the temperature and the

humidity of a *Plantation* asset every hour, and execute periodically the *asset_monitor* operation to record them on the SCM system. The asset_monitor operation consists of a sequence of checking operations that are performed at least once. The circular arrow means the iteration from 1 to n of the check operation.

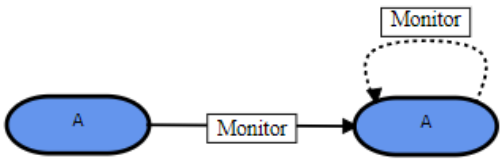


Figure 4.17: A basic example of asset_monitor() operation.

- *asset_compose* and *asset_decompose* (Figures 4.18 and 4.19): the compose operation creates a new asset using existing assets having the same owners without destroying them. The owner and the controller of the newly created asset are set to be the same as the originating assets. The decompose operation, instead, is used for tracing that a previously assembled asset has been disassembled, causing the original assets to be restored. The *asset_compose()* is somehow similar to the *asset_transform()* operation, because they both produce a new asset. However, the compose operation is revertible through the decompose one, while the transform operation is not revertible.

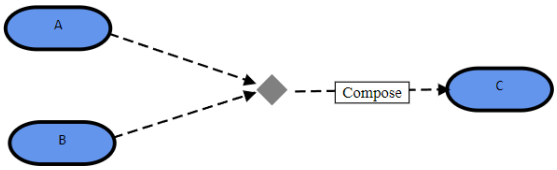


Figure 4.18: A basic example of asset_compose() operation.

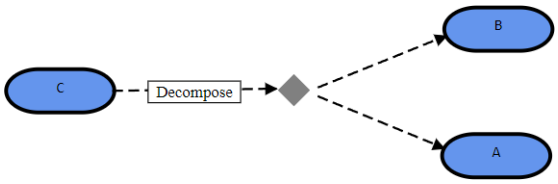


Figure 4.19: A basic example of asset_compose() operation.

- *asset_sell* (Figure 4.20): is a special transaction between two supply chain participants, *owner_a* and *owner_b*, representing that *owner_a* sells such

asset to *owner_b*. Hence, this operation changes the value of the owner property of an asset and, obviously, only the owner of an asset can perform the related *asset_sell* operation, specifying the id of the new owner. Symmetrically, in order to actually become the new owner of the sold asset, *owner_b* must explicitly accept the ownership of the asset by performing *asset_buy* operation. In the scGR, the *asset_sell* and the related *asset_buy* operations are represented by a red arrow between two assets having a label reporting the operation name: "Sell".

Figure 4.20: A basic example of asset_sell() operation.

- *asset_giveControl* (Figure 4.21): is another special transaction between two supply chain participants, *controller_a* and *controller_b*, representing the transfer of the control of an asset from *controller_a* to *controller_b*. Hence, this operation changes the controller property of an asset. Like all the other operations (but the *asset_sell* one), the *asset_giveControl* operation can be invoked only by the controller of the asset, who specifies the id of the supply chain participant that will be the new controller. Symmetrically, in order to actually become the new controller, *controller_b* must explicitly accept the control of the asset by performing an *asset_takeControl* operation. In the scGR, the *asset_giveControl* and the related *asset_takeControl* operations are represented by a red arrow between two assets having a label reporting the operation name: "giveControl".

Figure 4.21: A basic example of asset_giveControl() operation.

Please also notice that, although all the operations we have shown in this section are applied over consumable assets (and containers), the same operations are also applicable to non-consumable ones.

Our framework allows to pair a set of constraints to each operation. These constraints are mathematical or logical conditions computed on the properties of the two assets involved in the operation, and they must be satisfied in order the operation to be registered on the SCM system. In our reference example, a constraint could be defined on the harvesting operation (called *Plantation_Harvesting* in Section 2.4) which states that the ratio between the weight of the *Crop* asset that has been produced and the dimension of

the *Plantation* asset from which such crops have been produced must not exceed a given threshold.

4.1.4 Roles and Rights

Roles & Rights are used in our model to regulate the right of registering on the SCM system the execution of the previously described operations by the supply chain participants. In particular, our model adopts the Role-Based Access Control model (RBAC) [19] for defining authorization rules. Hence, when designing a supply chain through our tool, the supply chain domain expert at first creates a proper set of roles to be assigned to the supply chain participants, and then pairs an authorization rule to each of the operations of the supply chain to be protected. An authorization rule defines which role is required to be allowed to perform the operation is paired with. In the scGR, a light blue label attached to the arrow representing an operation represents the related authorization rule, and specifies which role is required to hold the right to perform such operation, as shown in Figure 4.22. Subsequently, when the smart contracts implementing the SCM system have been deployed on the blockchain, the supply chain administrator registers in the SCM system the unique addresses of the supply chain participants (e.g., the Ethereum address), and assigns to each of them a set of roles among the available ones (using the Administration Interface described in Section 7.1.1). Hence, the SCM system will allow the registration of the execution of an operation on an asset only to the supply chain participants holding the specific role requested for that operation (e.g., Farmer, Distributor, etc). Moreover, our framework imposes a further implicit authorization rule on all the operations, which states that, in order to be allowed to execute an operation on a given asset, a supply chain participant must be the controller of such asset. The only exception is for the *asset_ sell* operation, for which the implicitly authorization rule requires that the supply chain participant who wants to sell an asset must be the owner of such asset. The idea is that each operation executed on an asset requires a specific capability from the supply chain participant executing it, who must be also in control of such asset. For instance, the actor who plants *Seed* in a field must be a *farmer*, and must be the controller of the field. Hence, the SCM system must somehow check that the supply chain participant trying to register the execution of an operation was actually entitled to perform such operation, in order to protect the quality of the production process and hence of the final product.

Figure 4.22 shows the application of the role in the “Seed_Planting” operation. The specified operation can only be performed by a user whose associated role is “Farmer”: that is, only an user holding the role *Farmer* can perform the *Seed_ Planting* operation on the "Seed" asset at that precise step of the supply chain.

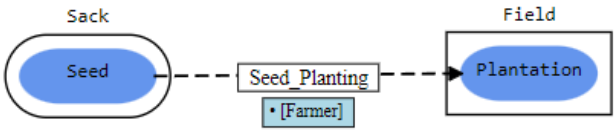


Figure 4.22: Example of operation role use, referring the snippet on Figure 4.23.

Notice that we chose to not associate specific capabilities when selling or transferring the control of an asset. For this motivation the *asset_sell* and *asset_giveControl* do not have associated RBAC authorization rule. Notice however that such operations are protected by two default authorization rules, that are always enforced by our framework, which grant the rights to perform the *asset_sell* and the *asset_giveControl* operations only to the owner and controller of the asset, respectively. Similarly, the *asset_buy* and the *asset_takeControl* operations are protected by other two default authorization rules which grant the right to perform such operations only to the blockchain participants that have been explicitly specified (as parameters) in the related *asset_sell* and the *asset_giveControl* operations, respectively.

4.2 Soy supply chain translation

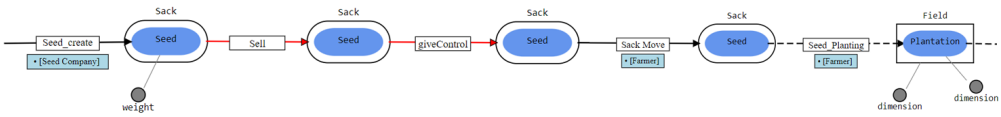


Figure 4.23: Part I of the soy bean supply chain, presented on Figure 2.4.

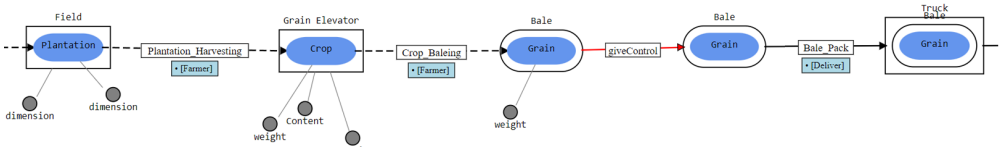


Figure 4.24: Part II of the soy bean supply chain, presented on Figure 2.4.

In this section we apply our model to represent the soybean supply chain described in Section 2.1.2. We suppose that the Soybeans Producer (*SBP*) wants to track the soybeans production process, from the seeds acquisition to the soybeans commercialization.

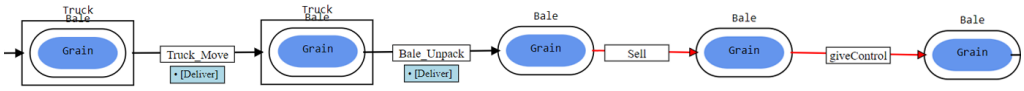


Figure 4.25: Part III of the soy bean supply chain, presented on Figure 2.4.

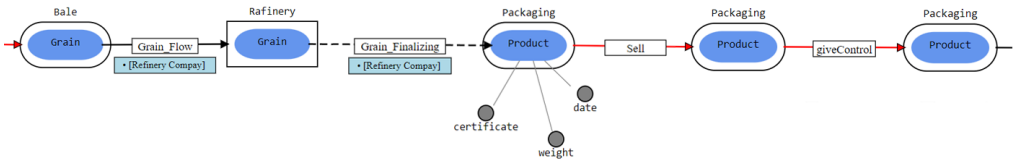


Figure 4.26: Part IV of the soy bean supply chain, presented on Figure 2.4.

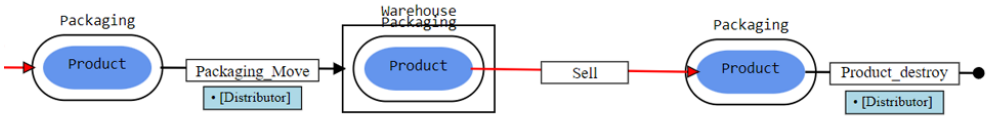


Figure 4.27: Part V of the soy bean supply chain, presented on Figure 2.4.

As the first step, the *SBP*, who is the supply chain domain expert, defines the assets and the operations of the soybean production process. Then, exploiting the tools of our framework, the *SBP* graphically represents the sequence of operations to be executed on the assets in the production process, and the skeletons of the smart contracts implementing the related SCM system are automatically produced. Figures 4.23 4.24 4.25 4.26 4.27 show a graphical representation (divided in five parts) of the soybean production process using our model.

The first asset of the supply chain is *Seed*, represented by the leftmost rounded box in Figure 4.23 The *Seed* asset is enclosed in a consumable container, called *Sack*, because it is a consumable and uncountable asset. A relevant property of the *Seed* asset is the weight, which represents the weight of the seeds in the sack. This asset has an incoming arrow which not originates from another asset, that in our framework assumes the meaning that such asset have no origin tracked in the system and that this asset is simply created by a participant of our SCM system through the asset_create operation (labelled in the scGR with the name *Seed_ create*), which also creates the *Sack* container for the *Seed* asset. As explained in Section 4.1, our framework adopts the RBAC model to regulate the rights to register the execution of operations on the SCM system. Hence, the SBP, when designing the soybeans supply chain through the scDI, can define the role(s) that is required to have the right to perform each of the operations in such supply chain. In

the reference example, the SCM system participants allowed to create *Seed* assets must hold the role *Seed Company*. This is represented in the scGR by the light blue label under the creation arrow.

The first operation performed on the *Seed* asset is the *Sell* one. Hence, a second instance of the *Seed* asset (enclosed in the *Sack* container as well) is present in the scGR on the right of the first one, and these two instances are connected through the arrow representing the *Sell* operation.

Since in the reference example we assume that the *Sell* operation also involves a physical transfer of the asset performed by the farmer (from the *Seed Company* premises to the farm), in the scGR the *Sell* operation is followed by a *giveControl* and then by a *Sack_Move* operation.

The *giveControl* operation is performed just after the *Sell* one, in order to register on the SCM system that another subject has taken the physical availability of the asset.

The *giveControl* operation must be executed before the *Sack_Move* one to change the controller of the *Seed* asset because our framework enforces the (implicit) authorization rule that operations on an asset can be performed only by the controller of such asset (with the exception of the *Sell* operation, as previously explained). For this reason, in the scGR there are other two instances of the *Seed* asset enclosed in the *Sack* container. The third instance is connected to the previous one through the arrow representing the *giveControl* operation, while the fourth instance is connected to the third one through the arrow representing the *Sack_Move* operation, which represents the physical transfer of the asset. The latter operation can be executed only by controllers holding the role *Farmer*, as shown in the scGR by the light blue label under the *Sack_Move* arrow, which contains the role name: *Farmer*.

The fourth operation in our reference example supply chain is the *Seed_Planting* one, which is performed on the *Seed* asset of a *Sack*, transforming it in a *Plantation* asset, which represents the specific area of the field where such seeds have been planted. Since the *Seed* asset and the *Sack* container are consumable, they are implicitly destroyed when the *Plantation* asset is created. Instead, the *Field* container embedding the *Plantation* asset is *non-consumable* container. Hence, in the scGR it is represented by a rectangle. Moreover, in our reference example we want that only users holding the role *Farmer* are authorized to execute the *Seed_Planting* operation. This authorization rule is represented in the scGR by the light blue label under the *Seed_Planting* arrow.

The fifth operation of our reference example supply chain is *Plantation_Harvesting*, which is executed on the *Plantation* asset (first operation shown in Figure 4.24). This operation can be executed only by entities having the role *Farmer*, as represented by the light blue label under the arrow

representing the *Plantation_ Harvesting* operation. The result of the *Plantation_ Harvesting* operation is the creation of a new asset, called *Crop*, which have a property, *weight*, representing its weight. The *Crop* assets are stored in a *Grain Elevator*, which is a *non-consumable* container having two properties: *capacity* and *content*. The first property, *capacity*, represents the maximum weight of crops that can be stored in the *Grain Elevator*, while the second property, *content*, represents the weight of the crops that are currently stored in the *Grain Elevator*. For the sake of this example, we define two constraints on the *Plantation_ Harvesting* operation. The first constraint states that the ratio between the weight of the *Crop* asset and the dimension of the *Plantation* asset from which it was produced must not exceed a given threshold. The second constraint states that the currently available capacity of the *Grain Elevator*, computed as the difference between the properties *capacity* and *content* of the *Grain Elevator* itself, must be greater than the weight of the *Crop* assets that the *Plantation_ Harvesting* operation declares that are being stored there. These constraints are defined by the SBP in the description of the Plantation_Harvesting operation, exploiting the edit panel of the scDI (see Section 5.1).

Several *Grain Elevator* containers could be defined, and the supply chain participant specifies the id of the *Grain Elevator* where the *Crop* assets have been stored among the parameters of the *Plantation_ Harvesting* operation.

The remaining operations of the soybeans supply chain are very similar the ones we have already described. For this reason, we are not providing a detailed description of Figure 4.25, Figure 4.26, and Figure 4.27.

We recall that the same production process could be represented in several distinct ways using our DSL, depending on which aspects of such production process the supply chain domain expert wants to highlight and hence trace with the SCM system.

CHAPTER 5

FROM SCJR TO SMART CONTRACT

This section describes the steps necessary for the automated translation of the graphic model represented in the scGR to the smart contract program code. We describe the integrated functions of the graphical interface, and how it produces a *json* document structured in such a way as to allow easy processing by a parser. The parser translates all occurrences of the *json* fields into smart contracts embedding proper data structures and methods, so that they can be easily used for deployment on a blockchain. In this thesis we focus on the Ethereum blockchain.

5.1 Supply Chain Design Interface

The supply chain Design Interface (scDI) is a graphical and user friendly web interface meant to allow its users, i.e., the supply chain domain experts, to represent the characteristics of their supply chains (as shown by step 1 of Figure 3.1), using the model and the DSL described in Section 4.1. A screenshot of the scDI is shown in Figure 5.1. This web interface shows in the central panel the graphical representation of the supply chain, and it provides four main functions, represented by the buttons on the left side of the window, which allow building the supply chain in terms of the four constructs defined in Section 4.1: Assets, Containers, Operations, and Roles & Rights.

The first function provided by the scDI is meant to add a new asset in the graphical representation of the supply chain shown in the central panel. Before creating the new asset by clicking on the Asset button, the

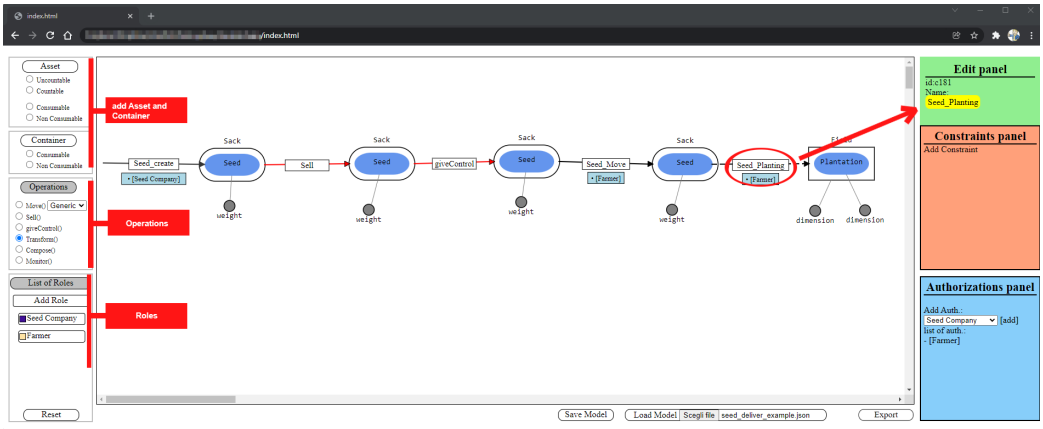


Figure 5.1: Example of Supply Chain Design Interface (scDI).

user must choose the asset characteristic, using the radio buttons under the Asset button: uncountable or countable, consumable or non-consumable.

Containers creation is the second basic function provided by the scDI. In this case, the user can choose among consumable and non-consumable containers. Notice also that, in our framework, containers are always countable. Once a container has been created (and it appears in the central panel), any of the asset objects already present in the panel (or any other container object with its relative content) could be dragged into it, to represent that such asset is stored in such container.

The operation construct defined in Section 4.1 is represented in our DSL by drawing an arrow that connects the two assets involved in the operation. In the scDI, the supply chain expert chooses the type of operation to be inserted (*move, transform, compose, monitor, giveControl* and *sell*) through the radio buttons under the Operations button. The drop down menu on the right of the move operation radio button allows to choose among the distinct types of move operations: *generic, update, pack or an unpack*. Then, by clicking on the Operations button, the supply chain expert can add an arrow connecting two assets and choose a unique name for that operation (except for *sell* and *giveControl* operations, which keep their original names).

Finally, the "List of Roles" panel allows the user to create the set of roles that are necessary for the specific supply chain. This list of roles is used to define the authorization rules to regulate the execution of operations on the SCM system. As we already pointed out, the only authorization rule imposed by default by the framework is that the sell operation can be executed on an asset only by the current owner of such asset, and all the other operations can be performed only by the current controller of the asset. As a matter of fact, the owner of an asset is the only entity entitled to sell the asset (i.e., to register on the SCM system that a sell operation has been executed), while

the controller of an asset is the only entity that can execute an operation (i.e., register on the SCM system the execution of an operation) on an asset, because the controller has the physical availability of the asset.

On the right side of the scDI there are three configuration panels: Edit panel, Constraints panel, and Authorizations panel:

- The Edit panel (the light green box on the top right of the scDI) allows the user to modify the names of assets, containers and operations, as well as to add properties to assets and containers.
- The Constraints panel (the light orange box in the middle right of the scDI) allows the user to set constraints on the execution of operations on assets. Such constraints are defined on the properties defined for the assets involved in the operations.
- The Authorizations panel (the light blue box on the bottom right of the scDI) allows the user to add authorization rules on the execution of operations on assets.

Clicking on an asset or on an operation gives access to the three previously mentioned panels. In these panels is possible to edit both assets and operations features: for assets, you can edit their name, set the point of creation and destruction, and add properties; as for operations, you can edit their name, add new constraints, and add a list of roles that are allowed to perform such operation. For instance, Figure 5.2 shows the edit panel of an asset named *Seed*. The Edit Panel allows user to change the name name of the asset, add the operation *Create()* (or *Destroy*) to the selected asset, add a property, and to clone the asset for a "reuse" of the specific asset in another step of the process. Similarly, Figure 5.3 shows the three panels of an operation named "Seed_Planting". The constraints panel is being used to set constraint "const_01". In particular, the constraints panel allows to choose which properties of the source asset and of the target asset are involved in the constraint among the available ones. Moreover, Figure 5.3 also shows that, through the Authorizations panel, the supply chain domain expert set that the role "Farmer" is required to be allowed to perform the operation "Seed_Planting"

Through the *Save Model* and the *Export* buttons (placed in the bottom of the scDI), two digital representations of the scGR drawn in the central panel are built: the scJR and the Solidity/web3js[1] script.

The *Save model* function exports the scJR into a file. The scJR file can be reloaded into the scDI through the *Load model* function, so that the supply chain previously saved can be modified (see step 2 of Figure 3.1). The function *Export*, instead, creates two different outputs: the solidity code

[1] `web3js.readthedocs.io/en/v1.3.4/`

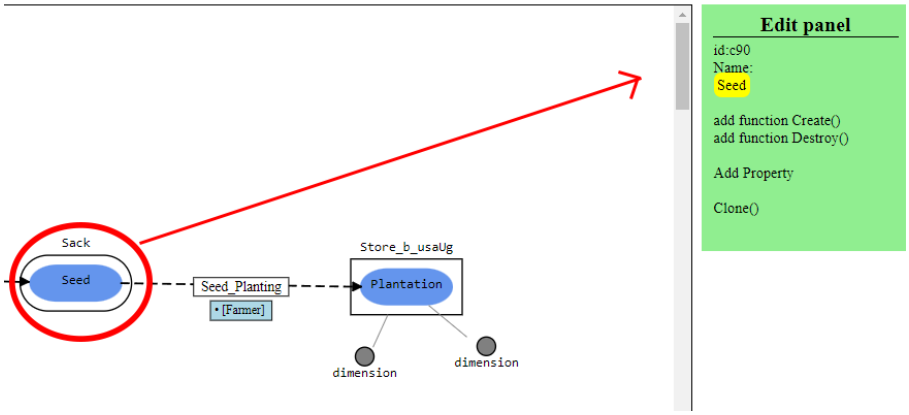


Figure 5.2: Example of the Edit panel of an Asset in the Supply Chain Design Interface (scDI).

and the "web3js interface" library for that code. In particular, the scJR produced using the scDI is translated into a *Solidity Skeleton* by the Supply Chain Model Translator, as shown in step 5 of Figure 3.1. Moreover, the scJR is also used by the Supply Chain Managing Interface Builder to create the Supply Chain Administration Interface, the Supply Chain Participant Interface, and the Supply Chain Viewer Interface (step 6 of Figure 3.1). In the following, we describe in details such components.

From the implementation point of view, the scDI consists of two JS files: the "init_graph.js" file defines the structure of the web page and the graphical representation, while the "main_index.js" file is a library. The first file initializes the graphical tool, loads the global variables, and uses the JointJs[2] package to build and manage the graphical aspect, while the basic functions of JQuery[3] are used to manage the resources for the framework. The second file describes the operations provided by the graphical environment, i.e., the operations defined by our model (see Section 4.1).

5.2 Supply chain JSON Representation (scJR)

The supply chain JSON Representation (scJR) produced using the scDI is a *JSON* formatted string consisting of six sections, namely:

- "*graph*",
- "*asset*",
- "*operation*",

[2] www.jointjs.com/
[3] jquery.com/

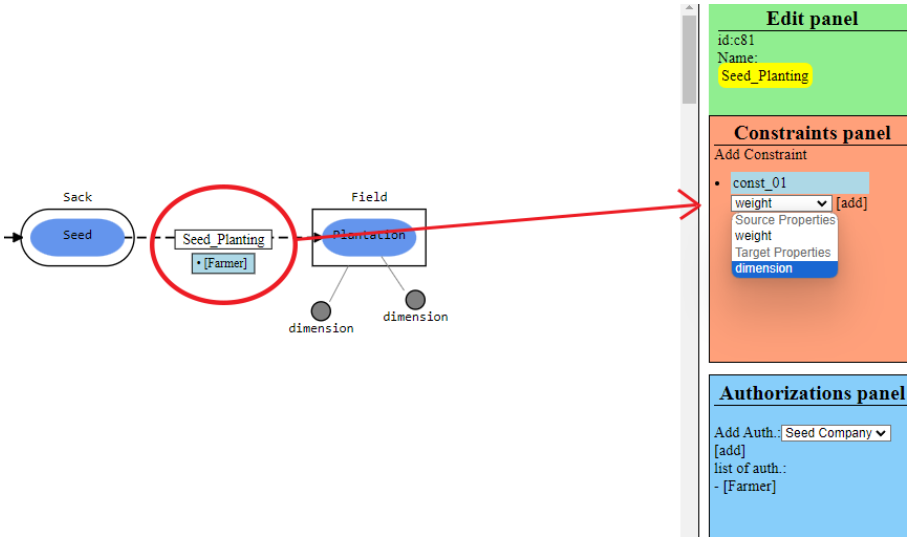


Figure 5.3: Example of the Edit panel of an Operation in the Supply Chain Design Interface (scDI).

- "*property*",
- "*constraints*",
- "*roles*".

The "*graph*" section gathers the declarations of all the objects needed by the framework's graphical libraries to rebuild the scGR.

The "*asset*" section lists all the assets as they appear in the scGR, specifying for each of them an *id*, the asset *name*, the *class*, i.e., asset or container, and the *type*. Type represent the type of asset among the four possible combination: 11 stands for uncountable and consumable asset; 12 stands for uncountable and non-consumable asset; 21 stands for countable and consumable asset; and 22 stands for countable and non-consumable asset. The "*operation*" section lists all the operations occurring in the scGR, i.e., all the arrows connecting couples of assets, specifying: name (taken fro the operation label on the scGR), class (defines the typology of the scGR entry), type (define the type of operation).

The "*property*" section lists all the properties related to each asset.

The "*constraints*" section lists the constraints paired with each operation.

The "*roles*" section saves all the roles that have been declared.

Figure 5.4 shows a simple example of scGR and Listing 1 shows a snippet of the related scJR. In the *asset* section of the scJR there are two entries (identified by "id":"c8" and "id":"c16") which represent the two states of the asset *A* in the supply chain; in the *operation* section there is an

entry (identified by "id":"c42") which represents the move operation having name *A_Move*, and other two entries representing the *A_Create* and the *A_Destroy* operations (identified by "id":"c54" and "id":"c38", respectively); in the *property* section there is one entry (identified by "id":"c23") representing the property called *property 01* of the asset identified by "id":"c8", i.e., the first state of asset *A*.

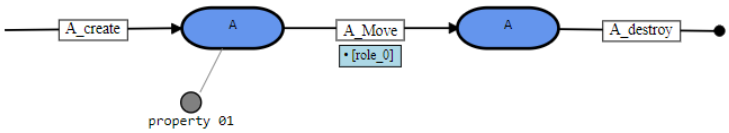


Figure 5.4: Example of Supply Chain Graphical Representation (scGR).

5.3 Smart Contracts Structure

Each of the smart contracts implementing the SCM system takes its name from the asset it represents, e.g., the smart contract "contract_A" represents the asset *A*.

In our model, assets having the same name in the scGR (and, consequently, in the scJR) represent the same asset in different moments of its life cycle, and hence they are represented with a single smart contract. Moreover, a further smart contract, *Accessrole*, is produced by extending the access permissions management library provided by the openZeppelin framework[4] (a built-in library written for solidity), to support the role-based access control on operations.

Each smart contract is composed of a data structure that stores the values taken by all the properties of all the instances of such asset in the states of their lifecycle, and of the set of functions that are invoked to keep trace on the SCM system of the operations that have been executed on the asset. A basic solidity code skeleton is shown in Figure 5.5: the basic structure of a solidity smart contract consists of a header, a section for variables, and a section for functions.

The solidity provides smart contract inheritance. Since it is defined as a high-level object-oriented language, we talk about context inheritance: variables and functions that a smart contract can inherit from other contracts. In the header, the name of the smart contract is declared. Also, it is specified here which other smart contracts are extended by this smart contract, i.e., from which smart contracts variables and functions are inherited. We use two classes to implement the RBAC protocol in handling transactions:

[4] `openzeppelin.com/contracts/`

```
{
    "graph":{},
    "asset":[
        {
            "id":"c8",
            "name":"A",
            "class":"asset",
            "type":"21"
        },
        {
            "id":"c16",
            "name":"A",
            "class":"asset",
            "type":"21"
        }
    ]
    "operation":[
        {
            "id":"c38",
            "name":"A_destroy",
            "class":"operation",
            "type":"destroy",
            "source":"c16",
            "target":{},
            "roles":[],
            "constraint":[],
            "selor":[},
        },
        {
            "id":"c42",
            "name":"A_Move",
            "class":"operation",
            "type":"move",
            "source":"c8",
            "target":"c16",
            "roles":[
                "role_0"
            ],
            "constraint":[],
            "selor":[},
        },
        {
            "id":"c54",
            "name":"A_create",
            "class":"operation",
            "type":"create",
            "source": {},
            "target": "c8",
            "roles":[],
            "constraint":[],
            "selor":[},
        }
    ],
    "property":[
        {
            "id":"c23",
            "name":"property 01",
            "fieldof":"c8"
        }
    ],
    "roles":[
        "role_0"
    ],
    "constraints":[
        {
            "id": "const_0_c23",
            "name": "const01",
            "constraint": "< 50",
            "properties": [
                "c31"
            ]
        }
    ]
}
```

Listing 1: scJR translation of the scGR in Figure 5.4

the *ERC20* class, which handles the creation of unique tokens, and the *AccessControl* class, which allows the implementation of useful functions for associating addresses to tokens, obviously generated with as ERC20. The superclasses ERC20 and AccessControl allow to declare unique labels, assign them to an address, and later to verify if an address holds one of such label.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.2 <0.9.0;

contract Contract_A {
    /*
      Variables section
    */
    uint256 number;

    /*
      Functions section
    */
    function store(uint256 num) public {    22520 gas
        number = num;
    }

    function retrieve() public view returns (uint256){    2415 gas
        return number;
    }
}
```

Figure 5.5: Example of solidity code structure.

We therefore used this library to declare role labels. In smart contract declaration, it is a good practice to divide the contract listing into two sections: one for the variables, and one for the functions. In the Variables Section, data structures, and enum sets are defined in addition to the variables that the smart contract needs. The Functions Section, instead, includes the constructor, the edit functions, the events, and the views functions.

Figure 5.6 shows a simple example of smart contract for the asset A derived from the scJR shown in Figure 1. The data structure consists of two dynamic nested arrays. The external array takes its name from the assets name (e.g., "store_A_s" for the asset A in the example of Figure 5.4) and it represents the collection of all the existing instances of the asset, each identified by its own (and distinct) ID. As a matter of fact, each entry of the external array is represented by a struct including the asset ID and an inner array, which represents the history of the instance, i.e., it stores all the states it has taken from its creation to its actual state. This array takes its name from the assets name, e.g. "A" for the asset A, and each entry is a structure storing the name of the state and the values of all the properties of the asset in that state, i.e., the owner ID, the controller ID, the asset physical position, and all the other user defined properties of the asset that are represented in the scGR (e.g., *property 01* in the example shown in Figure 5.4). The states that an asset can assume correspond to the execution of the operations defined on the asset itself, and they take their names from such operations with the postfix "_ed". Moreover, the initial state is called "Initialized" and the final state called "Destroyed". An alternative solution to implement the tracking of the operations executed on assets would have been to simply exploit the fact that the blockchain inherently tracks in its blocks the execution of smart contracts' functions. However, adopting this solution would have required to navigate back the blockchain blocks to find the ones embedding the transactions related to the asset we are interested

in, with higher computational cost. Moreover, the asset history would not have been visible from the smart contracts of our framework, but only from off-chain applications.

The operations defined on each asset in the scJR are implemented as functions of the smart contract representing such asset. These functions have the same name of the operations they represent, with the name of the asset as prefix. If the operation represents a change in the properties of the asset, the function of the corresponding smart contract updates the value of the variable representing such properties in the new asset state. If the operation, instead, represents the creation of a new asset (e.g., as consequence of the transformation of the original asset), then the corresponding function needs to invoke the smart contract representing the other asset to invoke the creation of the new instance. Further functions are always part of the smart contract representing an asset: one function for the creation of a new asset, one for destroying the asset, and the other for accessing the contents of an asset. These functions take their names from the assets name, e.g., "A_create()", "A_destroy()" and "A_view()" for the asset A.

The user defined constraints paired with the operations, i.e., the ones declared in the scGR, are implemented in the corresponding functions of the smart contract as "require()" commands. The default constraints imposed by the framework (e.g., the one requiring that user invoking the function must be the current controller of the asset) are implemented in the same way. The authorization rules are implemented through "require()" commands as well. In particular, these commands checks that the role of the user invoking the function is the one specified in the authorization rule in the scGR for the corresponding operation.

5.4 Supply Chain JSON Representation Translation into Smart Contracts

The Supply Chain Model Translator (scMT) is the tool of our framework in charge of producing the smart contracts implementing the SCM system starting from the scJR representing a supply chain (step 5 of Figure 3.1).

To generate the set of smart contracts building up the SCM system according to the above description, the scMT elaborates the scJR with three distinct procedures. The first procedure consists of a loop that parses all the entries of the *asset* section of the scJR, in order to identify all the distinct assets present in the supply chain. For each asset found, a distinct smart contract is created, following the structure described above. For instance, if we consider the scJR shown in Listing 1, which refers to the scGR in Figure 5.4, only one asset is found, asset A. Then, the other two procedures are ex-

ecuted to complete the code of such smart contracts: one for Properties, and one for Operations. The second procedure parses the entries of the *property* section of the scJR, in order to gather the set of properties related to each asset. Then, for each asset, say A, the declarations of a set of variables representing the set of properties previously identified are added in the declaration of the "asset_A_history" data structure of the related smart contract. The declarations of the variables representing the default asset properties (asset owner, controller and physical position) are added by the procedure in the "asset_A_history" data structure as well. As a matter of fact, since in the scGR in Figure 5.4 only one property, *property 01*, is paired with asset A.

The third procedure executed by the scMT parses the *operation* section of the scJR to determine the set of operations defined on each asset of the supply chain. All the operations have an unique id, identifying them in the set. Then, in the smart contract related to each asset, the scMT creates a function for each operation defined on such asset, taking the parameters to be passed to the function from the temporary nominal array of properties previously created.

In the following we describe how each section of the JSON file is translated for the definition of smart contracts elements. For simplicity we will use a simple supply chain for producing a smart contract, represented in figure Figure 5.4. In this figure we can see: two instances of an asset "A"; a property defined for asset, *property 01*; a generic update operation "A_update"; and a single role *role_0*.

5.4.1 Translation of Assets and Containers

Figure 5.4 shows a simple example of scGR and Listing 1 shows a snippet of the related scJR. In the *asset* section of the scJR there are two entries (identified by "id":"c8" and "id":"c16", see the excert in Listing 2) which represent the two states of the asset A in the supply chain; in the *operation* section there is an entry (identified by "id":"c42") which represents the move operation having name *A_Move*; in the *property* section there is one entry (identified by "id":"c23") representing the property called *property 01* of the asset identified by "id":"c8", i.e., the first state of asset A. Each smart contract is composed of a data structure that stores the values taken by all the properties of all the instances of such asset in the states of their life-cycle, and of the set of functions that are invoked to keep trace on the SCM system of the operations that have been executed on the asset. For each asset (let call it "<name>"), the declarations of a set of memory variables representing the set of properties previously identified are added in the "asset_<name>_history" data structure of the related smart contract. The declarations of the variables representing the default asset properties (asset owner, controller and physical position) are added by the procedure in the "asset_<name>_history" data

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity >= 0.8.0;
import "@openzeppelin/contracts/access/AccessControl.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract contract_A is ERC20, AccessControl {
    bytes32 public constant role_0 = keccak256("role_0");
    enum asset_states {Initialized, A_Move_ed,  Destroyed}

    struct asset_A_history {//properties
        string position;
        address Owner;
        address Controller;
        string property_01;
        asset_states state_of_A; //actual state
    }

    struct asset_A_struct{
        asset_A_history[] A;
        uint32 ID;
    }
    asset_A_struct[] public store_A_s; // MAIN STORAGE

    function A_Move(uint _ID, string memory position) public {    infinite gas
        uint len = (store_A_s[_ID].A.length)-1;
        require(store_A_s[_ID].A[len].state_of_A == asset_states.Initialized);
        require(store_A_s[_ID].A[len].Controller == msg.sender);
        require(hasRole(role_0, msg.sender), "ERROR: Function NOT Allowed!");
        // update the state array
        asset_A_history memory temp = store_A_s[_ID].A[len];
        temp.position = position;
        temp.state_of_A  = asset_states.A_Move_ed;
        // other code here ...
        store_A_s[_ID].A.push(temp);
    }

    function A_create(uint256 _ID, string memory position, string memory property_01) public {
        asset_A_history memory temp;
        temp.position = position;
        temp.property_01 = property_01;
        temp.Owner = msg.sender;
        temp.Controller = msg.sender;
        temp.state_of_A  = asset_states.Initialized;
        store_A_s[_ID].A.push(temp);
    }

    function A_destroy(uint256 _ID) public {    infinite gas
        uint len = (store_A_s[_ID].A.length)-1;
        require(store_A_s[_ID].A[len].state_of_A == asset_states.A_Move_ed);
        require(store_A_s[_ID].A[len].Controller == msg.sender);
        asset_A_history memory temp = store_A_s[_ID].A[len];
        temp.state_of_A  = asset_states.Destroyed;
        store_A_s[_ID].A.push(temp);
    }

    function A_view(uint256 _ID) public view returns(asset_A_history[] memory) {    infinite g
        return store_A_s[_ID].A;
    }

    // end of A's contract
    constructor()ERC20("name","SYM") {}    infinite gas 2780400 gas
}
```

Figure 5.6: Example of Solidity translation from the scJR.

structure as well. The procedure also creates a temporary nominal array which pairs each property with the functions where it appear as parameter. In addition, an "enum" label set is defined which represents the set of possible states of an Asset: the number and name of the possible state depends on the set of operations of the asset itself. Figure 5.6 shows a simple example of smart contract for the asset A derived from the scJR shown in Listing 1. The data structure consists of two dynamic nested arrays. The external array (or the "Main Dictionary") takes its name from the assets name (e.g., "store_A_s" for the asset A in the example of Figure 5.4) and it represents the collection of all the existing instances of the asset, each identified by its own (and distinct) ID. As a matter of fact, each entry of the external array is represented by a struct including the asset ID and an inner array, which represents the history of the instance, i.e., it stores all the states it has taken from its creation to its actual state. This array takes its name from the assets name, e.g. "A" for the asset A, and each entry is a structure storing the name of the state and the values of all the properties of the asset in that state, i.e., the owner ID, the controller ID, the asset physical position, and all the other user defined properties of the asset that are represented in the scGR (e.g., *property 01* in the example shown in Figure 5.4).

```
"asset":[
    {
        "id":"c8",
        "name":"A",
        "class":"asset",
        "type":"21"
    },
    {
        "id":"c16",
        "name":"A",
        "class":"asset",
        "type":"21"
    }
]
```

Listing 2: JSON representing the set of assets designed in the Figure 5.4

The same procedure used to produce the smart contracts related to assets is used for producing the smart contracts related to containers. Each container drawn in the scGR is represented in the scJR within the section "assets", and the type container is declared in the field "class". Unlike "assets", "containers" have implicitly declared and automatically implemented in the smart contract the create and destroy functions.

5.4.2 Translation of Operations

Since the supply chain described in the scGR in Figure 5.4 performs the sequence of operations A_create, A_Move, and A_destroy, the "operation" section of the scJR (see the excert in Listing 3) includes three entries, one for each operation, and the smart contract in Figure 5.6, includes four

```
1   "operation":[
2       {
3           "id":"c38",
4           "name":"A_destroy",
5           "class":"operation",
6           "type":"destroy",
7           "source":"c16",
8           "target":{},
9           "roles":[],
10          "constraint":[],
11          "selor":[],
12      },
13      {
14          "id":"c42",
15          "name":"A_Move",
16          "class":"operation",
17          "type":"move",
18          "source":"c8",
19          "target":"c16",
20          "roles":[
21              "role_0"
22          ],
23          "constraint":[],
24          "selor":[],
25      },
26      {
27          "id":"c54",
28          "name":"A_create",
29          "class":"operation",
30          "type":"create",
31          "source":"void",
32          "target": "c8",
33          "roles":[],
34          "constraint":[],
35          "selor":[],
36      }
37  ]
```

Listing 3: JSON representing the set of operations designed in the Figure 5.4

functions: A_Move (lines 24-35), and the default ones, i.e., A_create (lines 37-45), A_destroy (lines 47-54), and A_view (lines 56-58). The key solidity construct "require" [5] is inserted at the head of each function to check that a set of conditions are satisified before actually executing the function. The require construct generates an error message when the embedded condition is not satisfied. A condition that is added by default checks that, when the operation represented by this function is invoked, the current state of the asset is the one where this operation is defined, i.e., the operation is invoked at the the right step of the production process represented by the scGR. For all the operations (with the exception of the sell one) another condition that is added by default checks that the (address of the) blockchain user who invokes the function is (equal to the address of) the current controller of such asset. The address of the current controller of the asset is stored in the Controller variable of the current state of the asset itself. In case of sell operation, such condition, instead, checks that the (address of the) blockchain user who invokes the function is (equal to the address of) the current owner of such asset. The (address of the) current owner of the asset is stored in the Owner variable of the current state of the asset itself. In the smart contract in Fig-

[5] https://docs.soliditylang.org/en/v0.8.21/control-structures.html#panic-via-assert-and-error-via-require

ure 5.6, the two default conditions paired with the A_Move operation have been added at lines 26-27. Other conditions that the scMT procedure adds through the "require" statement to the functions of the asset smart contract concern authorization rules. In particular, when the scMT parses an entry of the *operation* section of the scJR, if the field *roles* contains a role, then a "require" statement that checks that the user invoking the operation holds such role is added to the function implementing such operation. We observe that the smart contract in Figure 5.6, for A_Move function, includes one authorization condition at line 28, because in the supply chain shown in Figure 5.4 the operation A_Move requires the controller to have the role *role_0* (light blue box under the operation arrow). Similarly, if the field *constraint* is not empty, the scMT procedure adds a further "require" statements implementing such constraint to the function in the asset smart contract representing such operation. The last step of the procedure adds the code implementing each function defined in each smart contract. This code depends of the type of the operation. For instance, in case of transform operations, the code in the body of the related function invokes the create function of the smart contract representing the new asset that is created by the transform operation. For the move operations, the position properties is updated with the new value passed as parameter. For the sell and give_control operations, instead, the body of the related functions updates the owner or the controller property, respectively, with the new address passed as parameter.

5.4.3 Translation of Properties

Each property drawn in the scGR is listed in section "property" of the scJR, and it is linked to the object (asset or container) it refers to via the "fieldof" field, where the static and unique id of such object is stored. Listing 4 shows an excert of the property section of Listing 1. In this particular instance, the property name "property 01" is linked to the asset "c8" declared in the asset subset.

During the creation phase of the smart contract related to an asset (or container), all properties that are related to such asset are searched in the "property" section of the scJR. In particular, since the same asset could be represented by multiple entries in the "asset" section of the scJR, all the ids of such entries must be taken into account when searching for the properties related to an asset. The resulting list of properties is added to the default ones in the history states structure of the smart contract. Hence, in our reference example, the "property 01" is added to the "asset_A_history" field structure of the smart contract produced for asset A.

```
"property":[
    {
        "id":"c23",
        "name":"property 01",
        "fieldof":"c8"
    }
]
```

Listing 4: JSON representing the set of properties designed in the Figure 5.4

5.4.4 Translation of Constraints

```
"constraints":[
    {
        "id": "const_0_c23",
        "name": "const01",
        "constraint": "< 50",
        "properties": [
            "c31"
        ]
    }
]
```

Listing 5: JSON representing the set of operation designed in the Figure 5.4

Constraints are represented in the scGR in each operation. Each constraint is defined on the properties defined on the assets involved in such operation: i.e., the properties defined on the asset on which the operation is executed and the properties defined on the asset resulting from the operation (that could even be the same asset). Listing 5 shows the "constraints" section of the scJR, where there is one entry defining one constraint. Each element has: a unique id (that is also reported in the field "constraints" of the operation it refers to); a name so that it can be represented in a new instance of the scGR; a list of the properties involved in the constraint. In Listing 5 the field *constraints* of the entry is paired with the string "$<$ 50", representing that the value of the property *c31* must be smaller than 50.

CHAPTER 6

SMART CONTRACT ANALYSIS

Chapter 5 describes how smart contracts are generated through the translating tool and their internal structure. In this chapter, we show a detailed analysis of both deployment and estimated execution costs of such smart contracts, in order to validate the *-chain framework, as well as the blockchain based SCM systems generated using the *-chain tools.

6.1 Smart Contracts Cost Analysis

This section evaluates the deployment and the execution costs of the smart contracts produced by our framework, starting from the given examples. First of all, it is important to observe that the cost of deployment of an SCM system strongly depends on the size and complexity of the related supply chain. The more assets will be present in the supply chain, the more smart contracts will be created by our framework and deployed on the blockchain, and the higher will be the SCM system deployment costs. Similarly, the more complex will be the production process, i.e., the more operations will be defined on an asset and/or the more properties will be required to represent the asset features, the most costly will be the deployment of the corresponding smart contract. Hence, in this section we evaluate the deployment cost of the assets of an SCM system varying the complexity, i.e., varying the number of properties paired with assets, varying the number of operations that can be executed on assets, and varying the number assets present in the supply chain.

6.1.1 Asset cost evaluation varying the number of properties

The first set of results we present in this section, shown in Table 6.1, is related to the SCM system obtained from the supply chain represented in Figure 5.4, varying the number of properties paired to asset A. Figure 6.1 shows an asset, A, having 10 properties, (called "property1", "property2", "property3"... "property10"), while Figure 6.2 shows the smart contract implementing such asset.

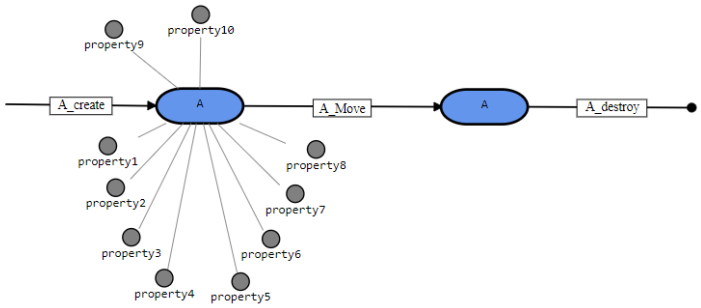


Figure 6.1: Example of Supply Chain Graphical Representation (scGR): move operation on an asset with ten properties.

Table 6.1 shows the cost of deploying the smart contract related to asset A and the cost of executing the two operations defined on such asset: A_create and A_Move. The costs are expressed in gas[1]. For what concerns the deployment cost, as expected, Table 6.1 shows that it increases relatively to the number of properties, and ranges from about 3546K (1 property) to 4782K gas (10 properties). It is important to notice that the deployment cost must be bore only once in the lifetime the SCM system. As a matter of fact, once the SCM system defined for a given supply chain has been deployed, it can be used to trace the related production process by simply creating new assets instances and invoking the operations defined for such assets, without further deployment costs. The cost of executing the A_create operation increases with the number of properties, and it ranges from 27K to 42K gas. This increase is motivated by the fact that the initial value of all the properties must be passed by the supply chain participant to the A_create function. The cost of executing the A_Move operation, instead, is almost constant, an it is about 25K gas. This is motivated by the fact that only one properties is updated in this case, i.e., the asset position, independently on the total number of properties of the asset. These cost are paid several time during the lifetime of the SCM system, i.e., every time that such operations are executed. We notice that the execution costs are two orders of magnitude

[1]https://ethereum.org/en/developers/docs/gas/

lower than the deployment cost. Finally, we also need to say that for each SCM system, independently on the number of assets and their properties or operations, additional smart contracts must be deployed for ERC20 tokens and for managing access control, having a cost of about 1246K for ERC20 and 83K gas units for the abstract interface contract.

#Properties	#Move Operations	Deployment Cost (gas)	Execution Cost of A_create (gas)	Execution Cost of A_Move1 (gas)
1	1	3545879	26688	25096
2	1	3585972	27753	25106
3	1	3741248	28995	25142
5	1	4045538	31372	25140
8	1	4502005	35189	25118
10	1	4781756	42103	25154

Table 6.1: Deployment costs and execution costs of the *A_create* and *A_Move* operations varying the number of properties of the asset *A*.

```
contract contract_A is ERC20, AccessControl {
    bytes32 public constant role_0 = keccak256("role_0");

    enum asset_states {Initialized, Move_ed, Destroied}

    struct asset_A_history {//properties
        string position;
        string property1;
        string property2;
        string property3;
        string property4;
        string property5;
        string property6;
        string property7;
        string property8;
        string property9;
        string property10;
        address Owner;
        address Controller;
        asset_states state_of_A; //actual state
    }

    struct asset_A_struct{
        asset_A_history[] A;
        uint32 ID;
    }
    asset_A_struct[] public store_A_s; // MAIN STORAGE

    function create_A(uint256 _ID, string memory position, string memory property1,
                      string memory property2, string memory property3, string memory property4,
                      string memory property5, string memory property6, string memory property7,
                      string memory property8, string memory property9, string memory property10) public {
        asset_A_history memory temp;
        temp.position = position;
        temp.property1 = property1;
        temp.property2 = property2;
        temp.property3 = property3;
        temp.property4 = property4;
        temp.property5 = property5;
        temp.property6 = property6;
        temp.property7 = property7;
        temp.property8 = property8;
        temp.property9 = property1;
        temp.property10 = property10;
        temp.Owner = msg.sender;
        temp.Controller = msg.sender;
        store_A_s[_ID].A.push(temp);
    }
```

Figure 6.2: Snippet of smart contract of Supply Chain Graphical Representation (scGR) of Figure 6.1

6.1.2 Asset cost evaluation varying the number of operations

The second set of results, shown in Table 6.2, are obtained varying the number of operations defined in the supply chain for the asset A.

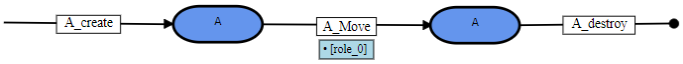


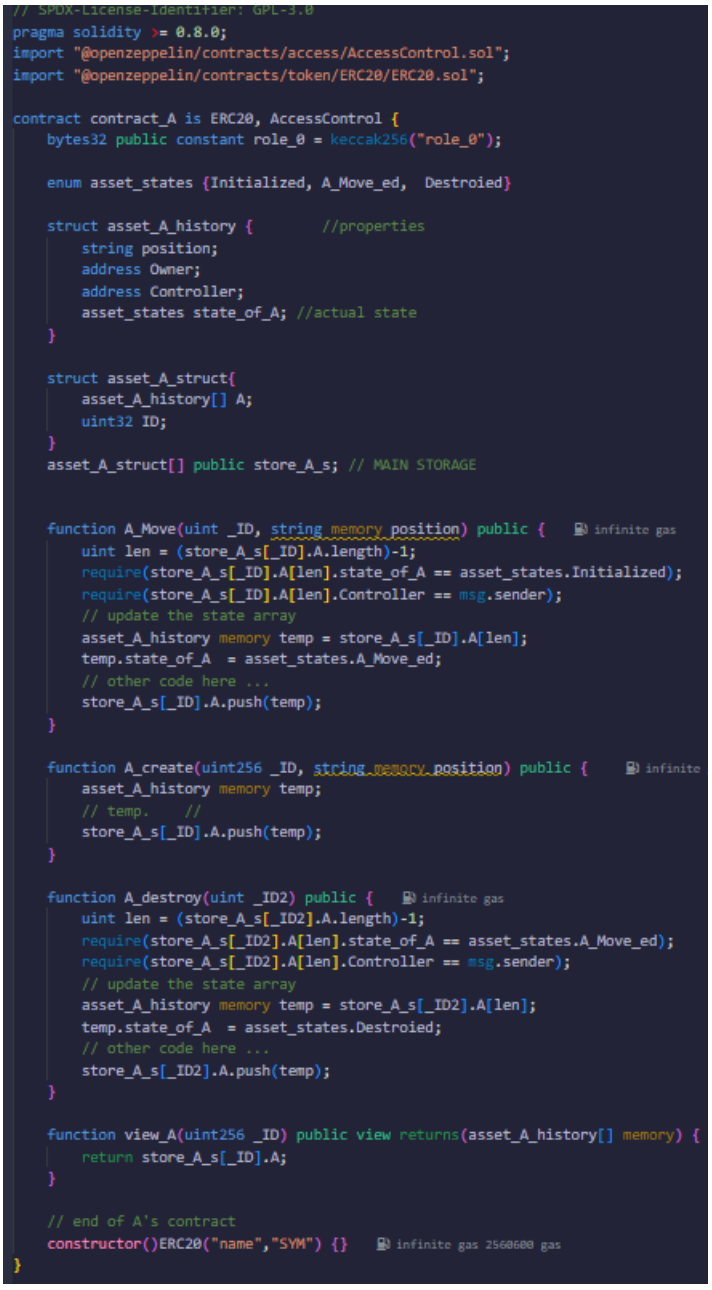
Figure 6.3: Example of Supply Chain Graphical Representation (scGR): single move operation.

Figure 6.3 shows a diagram in which an asset undergoes a move operation called "A_Move", and Figure 6.4 shows the corresponding solidity code auto generated by the diagram in Figure 6.3. This code contains only one user defined operation "A_Move" and the three default operations "A_create", "A_destroy" and "view_A".

Figure 6.5 shows a diagram in which an asset undergoes two possible move operations called "A_Move1" and "A_Move2". Hence, Figure 6.6 shows the corresponding solidity code auto generated by the diagram in Figure 6.5. this code embeds two user defined operations, respectively, "A_Move1" and "A_Move2", and the three default operations "A_create", "A_destroy" and "view_A".

Please notice that, although the *A_create* and the *A_Destroy* operations are always added by default by our framework, they are not included in the number of operation shown in the "#Move Operations" column of Table 6.2, which only refers to the moves operations. From the results reported in the first 5 rows of Table 6.2 taking account an asset A with one property only, we can observe that the deployment cost of the smart contract representing A increases when the number of operations defined for such asset increases, ranging from about 3546K gas (1 move operation) to about 5044K gas (5 distinct move operations). The costs for executing the *A_create* and *A_Move1* operations, instead, are constants and they are both about 27K and 25K gas, respectively, which are, again, two orders of magnitude less of the deployment cost. Moreover, the last row of Table 6.2 shows that, taking an asset with 3 move operations , if we increment the number of properties to 3, both the cost of deployment and the cost of execution of the *A_create* operation increase, while the cost of executing the *A_Move1* operation is almost constant, similarly to what observed in Table 6.1 for assets with 1 operation only.

Table 6.3 shows the results of a third set of experiments that have been conducted on supply chains similar to the one used for the second set of experiments, where move operations have been substituted with transform



```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >= 0.8.0;
import "@openzeppelin/contracts/access/AccessControl.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract contract_A is ERC20, AccessControl {
    bytes32 public constant role_0 = keccak256("role_0");

    enum asset_states {Initialized, A_Move_ed,  Destroied}

    struct asset_A_history {        //properties
        string position;
        address Owner;
        address Controller;
        asset_states state_of_A; //actual state
    }

    struct asset_A_struct{
        asset_A_history[] A;
        uint32 ID;
    }
    asset_A_struct[] public store_A_s; // MAIN STORAGE


    function A_Move(uint _ID, string memory position) public {    infinite gas
        uint len = (store_A_s[_ID].A.length)-1;
        require(store_A_s[_ID].A[len].state_of_A == asset_states.Initialized);
        require(store_A_s[_ID].A[len].Controller == msg.sender);
        // update the state array
        asset_A_history memory temp = store_A_s[_ID].A[len];
        temp.state_of_A  = asset_states.A_Move_ed;
        // other code here ...
        store_A_s[_ID].A.push(temp);
    }

    function A_create(uint256 _ID, string memory position) public {    infinite
        asset_A_history memory temp;
        // temp.    //
        store_A_s[_ID].A.push(temp);
    }

    function A_destroy(uint _ID2) public {    infinite gas
        uint len = (store_A_s[_ID2].A.length)-1;
        require(store_A_s[_ID2].A[len].state_of_A == asset_states.A_Move_ed);
        require(store_A_s[_ID2].A[len].Controller == msg.sender);
        // update the state array
        asset_A_history memory temp = store_A_s[_ID2].A[len];
        temp.state_of_A  = asset_states.Destroied;
        // other code here ...
        store_A_s[_ID2].A.push(temp);
    }

    function view_A(uint256 _ID) public view returns(asset_A_history[] memory) {
        return store_A_s[_ID].A;
    }

    // end of A's contract
    constructor()ERC20("name","SYM") {}    infinite gas 2560600 gas
}
```

Figure 6.4: Snippet of auto-generated solidity code of the move operation in Figure 6.3

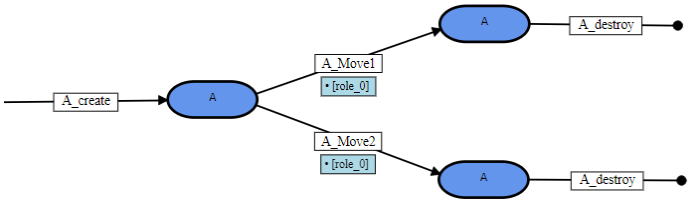


Figure 6.5: Example of Supply Chain Graphical Representation (scGR): two move operations.

#Properties	#Move Operations	Deployment Cost (gas)	Execution Cost of A_create (gas)	Execution Cost of A_Move1 (gas)
1	1	3545879	26688	25096
1	2	3920498	26503	25118
1	3	4295137	26559	25140
1	4	4669597	26585	25170
1	5	5044301	26550	25196
3	3	4664381	28947	25130

Table 6.2: Deployment costs and execution costs of the *A_ create* and *A_ Move1* operations varying the number of move operations defined on the asset *A*.

operations. The new assets that are created by the transform operations have one property only. As for the previous experiments, we evaluated the cost of deploying the smart contract related to asset *A* and the costs of executing the asset creation function, *A_ create*, and a transform operation, *A_ Transform1*, varying the number of transform operations defined for such asset. From the results shown in Table 6.3 we observe that the trend is similar to the one shown in Table 6.2 for the move operation. As a matter of fact, the smart contract deployment cost increases with the number of transform operations defined on the asset, while the execution costs of the *A_ create* and of the *A_ Transform1* operations is almost constant when the number of transform operations defined on the asset changes. Moreover, the last row of the Table shows that, if we increase the number of properties defined for asset *A*, the cost of the *A_ create* operation increases, while the cost of executing the *A_ Transform1* operation is not affected.

6.1.3 SCM System cost varying the number of assets in the supply chain

Finally, the last set of experimental results evaluate the deployment cost of supply chains consisting of a sequence of n transform operations which, from

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >= 0.8.0;
import "@openzeppelin/contracts/access/AccessControl.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract contract_A is ERC20, AccessControl {
    bytes32 public constant role_0 = keccak256("role_0");

    enum asset_states {Initialized, A_Move1_ed, A_Move2_ed,  Destroied}

    struct asset_A_history {        //properties
        string position;
        address Owner;
        address Controller;
        asset_states state_of_A; //actual state
    }

    struct asset_A_struct{
        asset_A_history[] A;
        uint32 ID;
    }
    asset_A_struct[] public store_A_s; // MAIN STORAGE


    function A_Move1(uint _ID, string memory position) public {    infinite gas
        uint len = (store_A_s[_ID].A.length)-1;
        require(store_A_s[_ID].A[len].state_of_A == asset_states.Initialized);
        require(store_A_s[_ID].A[len].Controller == msg.sender);
        // update the state array
        asset_A_history memory temp = store_A_s[_ID].A[len];
        temp.state_of_A  = asset_states.A_Move1_ed;
        // other code here ...
        store_A_s[_ID].A.push(temp);
    }

    function A_Move2(uint _ID, string memory position) public {    infinite gas
        uint len = (store_A_s[_ID].A.length)-1;
        require(store_A_s[_ID].A[len].state_of_A == asset_states.Initialized);
        require(store_A_s[_ID].A[len].Controller == msg.sender);
        // update the state array
        asset_A_history memory temp = store_A_s[_ID].A[len];
        temp.state_of_A  = asset_states.A_Move2_ed;
        // other code here ...
        store_A_s[_ID].A.push(temp);
    }
    function A_create(uint256 _ID, string memory position) public {    infinite gas
        asset_A_history memory temp;
        // temp.    //
        store_A_s[_ID].A.push(temp);
    }

    function A_destroy(uint _ID2) public {    infinite gas
        uint len = (store_A_s[_ID2].A.length)-1;
        require(store_A_s[_ID2].A[len].state_of_A == asset_states.A_Move1_ed
                || store_A_s[_ID2].A[len].state_of_A == asset_states.A_Move2_ed);
        require(store_A_s[_ID2].A[len].Controller == msg.sender);
        // update the state array
        asset_A_history memory temp = store_A_s[_ID2].A[len];
        temp.state_of_A  = asset_states.Destroied;
        // other code here ...
        store_A_s[_ID2].A.push(temp);
    }

    function view_A(uint256 _ID) public view returns(asset_A_history[] memory) {
        return store_A_s[_ID].A;
    }

    // end of A's contract
    constructor()ERC20("name","SYM") {}    infinite gas 2831400 gas
}
```

Figure 6.6: Snippet of auto-generated solidity code of the move operation in Figure 6.5

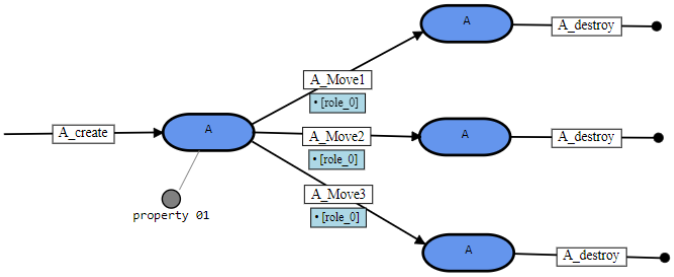


Figure 6.7: Example of Supply Chain Graphical Representation (scGR): multiple operations.

#Properties	#Transform Operations	Deployment Cost (gas)	A_create Cost (gas)	A_Transform1 Cost (gas)
1	1	3199521	26617	26218
1	2	3253698	26615	26218
1	3	3409800	26603	26218
1	4	3565987	26693	26218
1	5	3722093	26671	26218
3	3	3837147	28914	26218

Table 6.3: Deployment costs and execution costs of the *A_ create* and *A_ Transform1* operations varying the number of transform operations defined on the asset *A*.

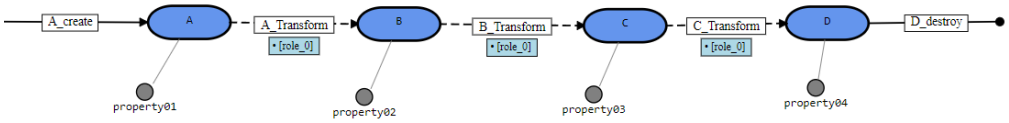


Figure 6.8: Example of Supply Chain Graphical Representation (scGR): sequence of transform operations

an initial asset, produce a number of intermediate ones and then produces the final assets. Figure 6.8 shows an example performing a sequence of 3 transform operations which involve 4 assets. Table 6.4 shows the total costs in gas of the related SCM systems, which include the costs of the smart contracts representing all the assets of the supply chain and of the smart contracts required for managing access control. As expected, the cost increases relatively as the number of assets in the supply chain increases.

#Assets	SCM System Deployment Cost (gas)
2	7708450
3	10925263
4	14141651

Table 6.4: Total deployment cost of a SCM system representing a sequential supply chain varying the number of assets.

CHAPTER 7

SCMS GRAPHICAL INTERFACES

This section shows the three auto-generated graphical interfaces from the graphical environment translation and SC interface builder tool. The three interfaces represent the nodes with which three different types of users interact at the SCM system, such as are administrator, participant, and viewer (the products' final customers). Each interface has different functions and purposes of use, but all three exploit the codes generated by the smart contract construction tool, i.e. all the functions implemented in the graphical interfaces are exactly the functions described in the generated smart contract.

7.1 Application Interfaces

As described in Section 3.1, the *-chain framework creates and provides to its users three user-friendly graphical environments: *i)* the Supply Chain Administration Interface, *ii)* the Supply Chain Participant Interface, and *iii)* the Supply Chain Viewer Interface.

In this chapter we describe these interfaces: their main functions, and how they interact with smart contracts. Moreover, we also analyze the procedure to deploy such interfaces in a web-oriented system in order to be used by the SCM system users (supply chain participants and final customers).

7.1.1 Supply chain Administration Interface

The supply chain Administration interface (scAI), shown in Figure 7.1, is automatically generated by the export function of the scDI, as shown in

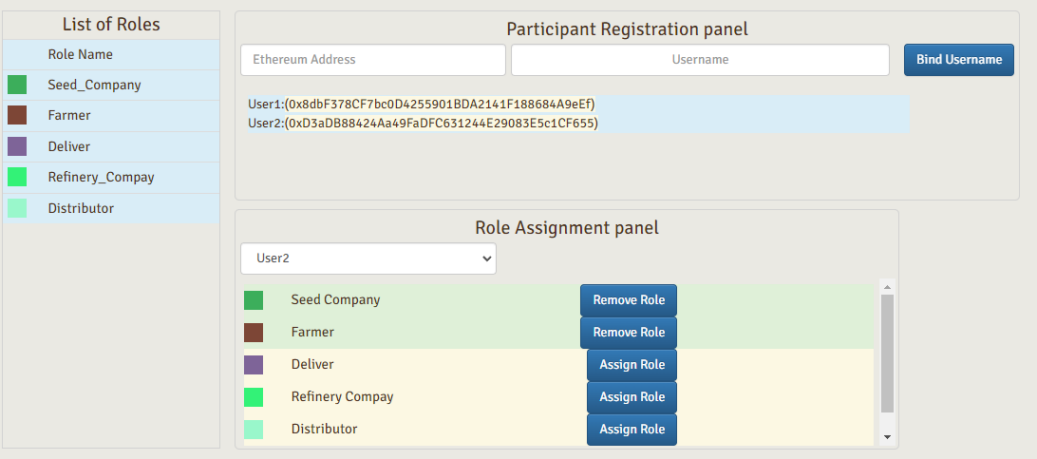


Figure 7.1: Example of Supply Chain Administrator Interface (scAI).

Figure 3.1 (step 6).

The scAI allows the supply chain administrator to register blockchain users as participants to the supply chain, and hence to the SCM system, and to assign and revoke their roles, in order to give them the rights to register on the SCM system the operations they perform on assets. The scAI consists of three sections: List of Roles, Participant Registration panel, and Role Assignment panel. The List of Roles section, placed on the left side of the scAI, lists all the roles that have been defined when the supply chain has been designed through the scDI. The Participant Registration panel is used by the supply chain administrator to associate the Ethereum addresses of users, with their human-readable names. The supply chain administrator knows such users, that are the ones which will operate on the supply chain. The *Bind Username* button creates the association between the address and the string representing the user name by invoking the create_user function of the web3js interface library. Since the names of supply chain participants are personal data, they are not directly stored on the blockchain. Instead, they are saved in an off-chain storage, and the related pointer is saved in the blockchain.

In the Role Assignment panel of the scAI, the registered users are listed in a drop-down menu. By selecting a username, the supply chain Administrator can assign one (or more) of the available roles to that user. Since the supply chain administrator knows the supply chain participant, they can assign them the right roles. By pressing the button *Assign Role* on the right of a role name, the function set_role of the Accessrole smart contract is called in order to assign that role to the selected user. By pressing the button *Remove Role*, instead, the function revoke_role of the Accessrole smart contract is used in order to delete a previous role assignment.

For instance, in Figure 7.1, the supply chain administrator registered two users, called "User1" and "User2", and assigned to the former the roles "Seed Company" and "Farmer".

7.1.2 Supply chain Participant Interface

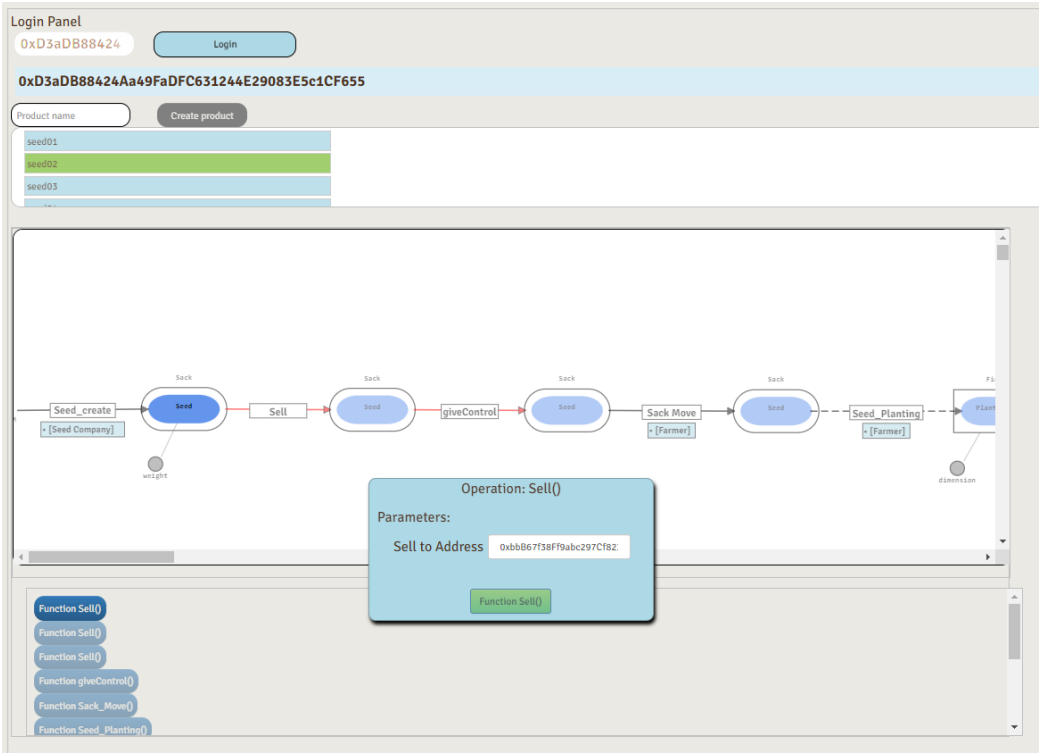


Figure 7.2: Example of Supply Chain Participant Interface (scPI).

The supply chain Participant Interface (scPI), shown in Figure 7.2, is automatically generated by the export function invoked from the scDI, as shown in Figure 3.1 (step 7).

This web interface is meant to be used by supply chain participants to claim the execution of the operations on the assets, according to the defined supply chain and to the roles they have been assigned by the supply chain administrator. The scPI consists of one main panel which shows the list of assets of which the supply chain participant is the current controller or the owner. By selecting one of these assets, a simplified version of the related scGR is shown in the central panel, while in the bottom panel the scPI shows the list of operations that the participant has the right to perform on the selected asset, according with the participant's role and the current status of the asset.

Figure 7.2 shows the “Login Panel” with an example concerning the user having Ethereum address "0xD3aDB88424Aa49FaDFC631244E29083E5c1CF655", which has been previously registered as “User2”. The assets shown in the top panel are those for which User2 is the controller or the owner. By selecting one of listed assets, the scPI shows a set of buttons that can be used to claim the execution of the available functions, according to the role held by User2. As assigned in the scAI, shown in the Figure 7.1 the User1 user holds the "Seed company" and the "Farmer" roles. Since we selected the asset “Seed02”, the central panel shows an excerpt of the related scGR where such asset is placed, while the bottom panel of the scPI shows a button which allows User2 to claim the execution of the *Sell()* operation on this asset, which is the only operation that can be executed on that asset in its current state. By pressing the "Function Sell()" button, the supply chain participant declares to have executed the *Sell()* operation on the asset they selected: a modal input form then appears in the center of the scPI, in which the user can set parameters to be provided to the operation. In this specific case, the selected function is a *Sell()* operation that requires the address of the new owner as parameter. The request operation is then sent to the blockchain system via the web3js library, awaiting for the callback answer notifying the correct execution of the operation.

7.1.3 Supply chain Viewer Interface

The supply chain Viewer Interface (scVI) is meant for the final customers, to allow them to inspect the production process which generated the product they are buying or consuming. The viewer interface is a simple web interface where the user provides the identifier of the product they want to know the production history. In real world applications, such identifier could be represented as a QR Code to simplify the fruition of the viewer interface. The web interface is divided into three sections. In the first section, on top of the interface, there is the input field where the identifier of the product to be inspected must be inserted. Such identifier embeds both the address of the smart contract implementing such kind of product and the ID of the specific product instance we are looking for. The ID is searched in the smart contract history, thus returning the last state which such specific asset instance is in, and providing the related information (e.g., the current values of the properties). In the central section of the scVI, the graphical representation of the supply chain involving such asset instance is automatically drawn, where all the elements are displayed as transparent blocks, except for the one representing the current state of the asset instance. Finally, in the bottom part of the scVI, the list of the operations that have been performed on each of the assets involved in the supply chain is shown. The operation that has

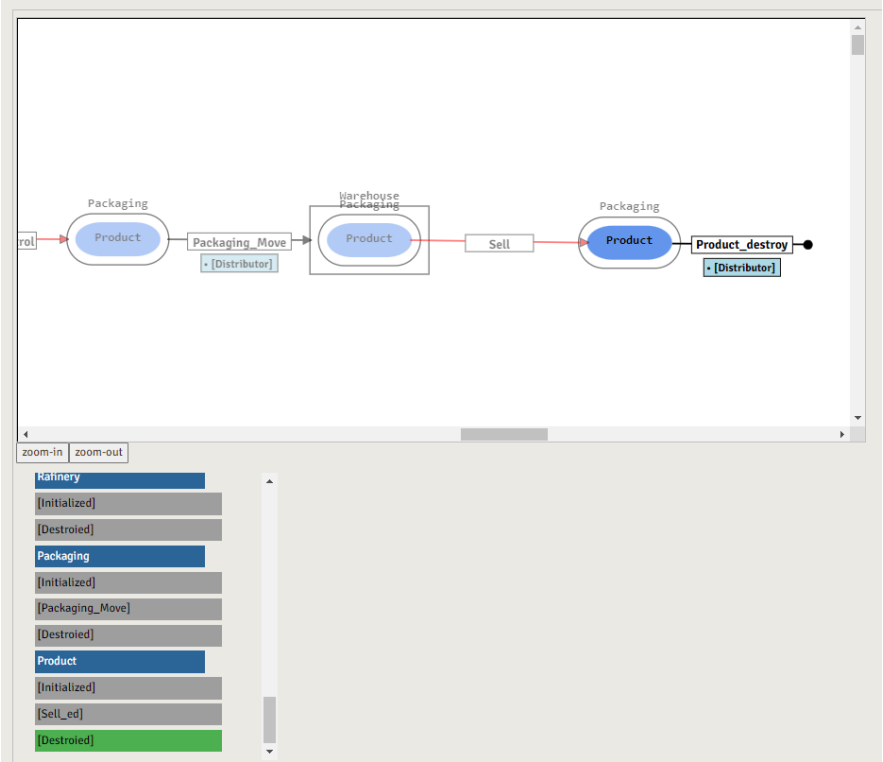


Figure 7.3: Example of she supply chain Viewer Interface (scVI).

been executed on the asset instance we are inspecting is highlighted in green. The other operation that would have been possible to execute on the asset instance in that state are shown in the list as well (not highlighted), while all the other operations defined on such kind of asset (but not possible in the state of the asset instance we are inspecting) are shown in light grey.

Figure 7.3 shows a snippet of the graphical interface in which the diagram represents the final step of a supply chain, where the product is sold to the final customer.

7.2 Application Deployment

In this section we address in detail how the previously described interfaces are loaded into web systems geared for use by users. Figure 7.4 describes the workflow going into details. In particular, we describe the steps required for the operation of the web-oriented application, we analyze which systems and technologies are involved in the deploying process, both at web server side and at blockchain network side.

Each of the previously described web interfaces must be available on a web service infrastructure. In our specific implementation, most of the code

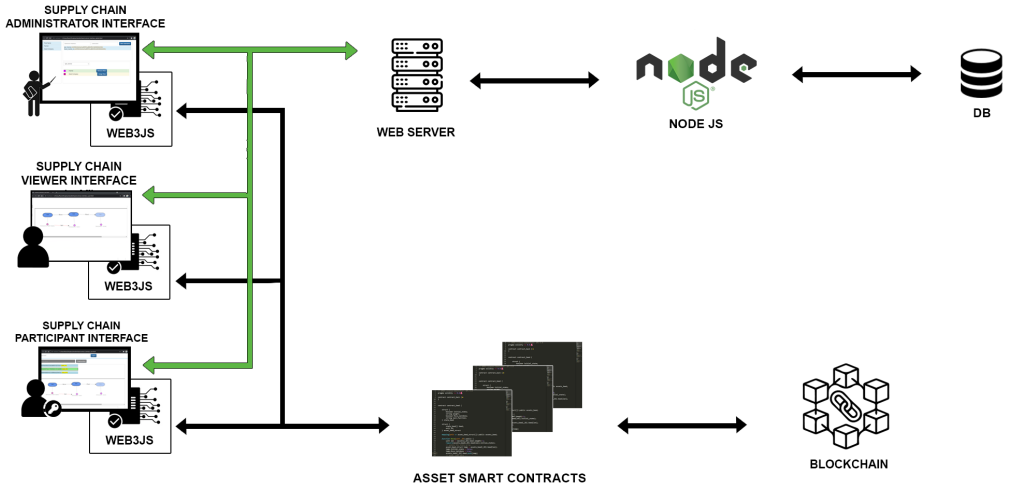


Figure 7.4: *-chain Framework usage workflow.

consists of JavaScript and JavaScript script-based frameworks. Ourchoice was to build a docker architecture sharing the interface, in order to keep divided the client interfaces to the web services, hence a Node.js service created in a distinct container: for each of the previously generated interfaces is implemented an individual container.

We used Node.js 18.16.0 [1] as base for the web service, also we do not use any particular additional modules except of "server.io" and the common local databases interface functions. As local database, we chose a standard mongoDB [2] with a common configuration to ensure simple json data storage. The local data base is needed to grant user authentication, managing sensitive data, and storing the tool configuration.

At its core, each auto-generated interface contains all the components to run in any web service. However, it is possible to very easily configure the tools in the interface so that they can be encapsulated in a larger or different application than the one proposed automatically by the framework.

Each interface, loads a common JavaScript module, where the functions required to communicate with the smart contracts implmenting the SCM system are exposed. This file is called "contract_interface.js" and is produced when the three interfaces are produced. This file, however, must be customized before being used by the interfaces. In particular, in the header of the file, there are four global variables that must be set with the addresses of the smart contracts implementing the SCM system. Therefore, only once the smart contracts has been deployed in blockchain, the four variables should be

[1] `https://nodejs.org/en`
[2] `https://www.mongodb.com`

set. The four variables involved are: the address of the blockchain node you want to connect to in order to invoke the functions; the blockchain address of the reference smart contract; the json that encapsulates the functions of the smart contract (called the Application Binary Interface, or ABI); and the blockchain address of the user or the entity who invokes the function.


```
[
  {
    "inputs": [
    ],
    "stateMutability":"nonpayable",
    "type":"constructor"
  },
  {
    "anonymous":false,
    "inputs": [
      {
        "indexed":true,
        "internalType":"address",
        "name":"owner",
        "type":"address"
      },
      {
        "indexed":true,
        "internalType":"address",
        "name":"spender",
        "type":"address"
      },
      {
        "indexed":false,
        "internalType":"uint256",
        "name":"value",
        "type":"uint256"
      }
    ],
    "name":"Approval",
    "type":"event"
  },
  {
```


Figure 7.5: A partial snippet of the generated ABI of the relative smart contract of the smart contract code of Figure 6.4

Figure 7.5 shows a cross-section of the *ABI* file, which is required for communication between web interfaces and the smart contracts residing in the blockchain: for the sake of visualization, the entire file is not shown because it would be difficult to understand. Such file is a json document containing at each occurrence all the start contract functions, both user-declared functions and functions inherited from the *ERC20* and *AccessControl* classes.

CHAPTER 8

TWO USE CASES

This Chapter shows two relevant and real use cases where the *-chain has been successfully applied to model the related supply chains and to produce the blockchain based SCM system as well as the graphical interfaces for interacting with it. The first use case concerns the supply chain for the production of olive oil while the second use case concerns the supply chain for producing a specific kind of meat. In both cases the supply chain must follow a specific regulation to obtain a quality certification for the final product.

8.1 Use case 1: PDO Olive Oil

The traceability of the production processes behind agri-food products is of paramount importance to witness their quality and to justify potentially high prices. To this aim, Protected Designation of Origin (PDO) labels for agricultural products is meant to ensure that the related production process had took place in specific region, following specific constraints and according to specific quality standards. For instance, *Terre di Siena* [16] is one of the PDOs registered for olive oils on the eAmbrosia[1] European legal register of the names of agricultural products and foodstuffs, wine, and spirit drinks. Just to give and example of the constraints imposed by the production regulation, the olives must originate from at least two of the following cultivars present, at commercial farm level, singularly for at least 10% and jointly to an extent of no less than 85%: Frantoio, Correggiolo, Leccino and Moraiolo. Such olives must be produced in the hilly territories of the province of Siena

[1] https://ec.europa.eu/info/food-farming-fisheries/food-safety-and-quality/certification/quality-labels/geographical-indications-register/

(in Tuscany, Italy). The harvest process can begin in October and must terminate by the end of December. The healthy olives must be kept in special fresh and ventilated premises for no more than 3 days from picking to avoid overheating and fermentation. Such olives are ground only with physical and mechanical methods, in oil-mills situated in the production territory, within 24 hours from their conferral to the oil-mill. [14] describes the European regulation for the PDO nomenclature, both for specific local products and for manufacturing types of production. From a practical point of view the oil production consists of several processing stages to which the olives are subjected after harvesting.

However, normative and regulations such as the previously mentioned PDO, are of limited use if a proper and trusted infrastructure for their enforcement are not put in place. In the light of these considerations in the following we describe how the *-chain framework can be used to design and implement a SCM system for tracing the olive oil production process.

The reference production process of an high quality olive oil we take into account in the following of this paper begins in the cultivation field. After the harvesting, the olives are transported to the mill. In the mill the olives are prepared for processing. This phase includes different operations: husking (elimination of foreign material), selection (division according to the level of health and size), and washing the olives. Depending on the technology used by the company, these operations can be conducted together or separately. We define four different phases regarding the definition of rules and constraints following the European regulations (shown in Figure 8.1): the *Preparation* phase, the *Crushing* phase, the *Oil Gathering* phase, and the *Bottling* phase.

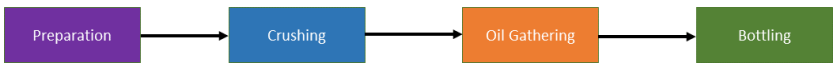


Figure 8.1: The main four phases of Olive Oil PDO Supply Chain

In the following of this chapter we will describe how each of such phases can be represented using the *-chain framework.

8.1.1 PDO Olive Oil Supply Chain Model

The Preparation phase

The operations performed in the preparation phase are shown in Figure 8.2. First of all, to prepare the olives for processing, a set of operations must be performed, i.e., weighing, selection (according to the health status, size and cultivar type), and olives wash. These operations are represented in the Harvesting lot step. After the harvesting, olives are gathered into proper

crates, which identify the olive lot, and they are conserved for a short period of time, maximum 24h (Conservation step in Figure 8.2). After conservation, some olive lots are used for the next steps (Reuse), some lots are sold (Sale), while other lots are given to other actors of the supply chain to continue the production process (Conferral). The olive lots are then moved to the oil mill, they are weighed and husked, and then they are ready for the delivery step).

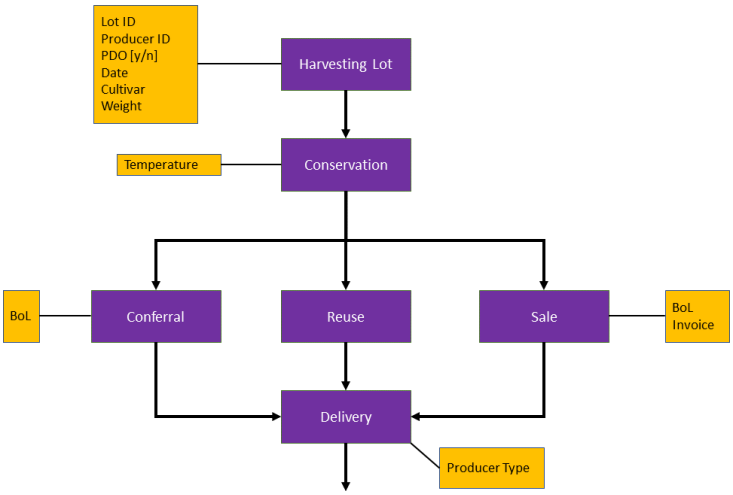


Figure 8.2: A snippet of PDO Olive oil Supply chain (I): the Preparation phase

The Crushing phase

In this phase, the Pressing operation is performed, where the cell wall is broken to extract the olives juice and paste. In particular, the break is caused in partially by the action of the chosen pressing system (impacts, rotation, or slashing) and in partially by the mechanical action of the core fragments. The Pressing operation is performed using traditional mills or mechanical crushers.

Moreover, the processing can thus take place in a very short time, and with a minimum footprint (especially if compared to the mullers). The crushers can be: discs, hammers, knives, and the Pitting machine.

In the kneading phase, the oil drops from the aggregate paste produced in the crushing phase. This operation extracts the greater volume of oil of the entire process. Even if most of the "traditional milling systems" use heavy millstones (which operate a shatters-kneading), modern productions use special machinery, called "malaxers": these machinery are continuous systems, since they do not require a long stop between the different operations. During this technological process, the paste coming from the pressing of the olives is slowly stirred. In this way, the breakage of the oil-water emulsion

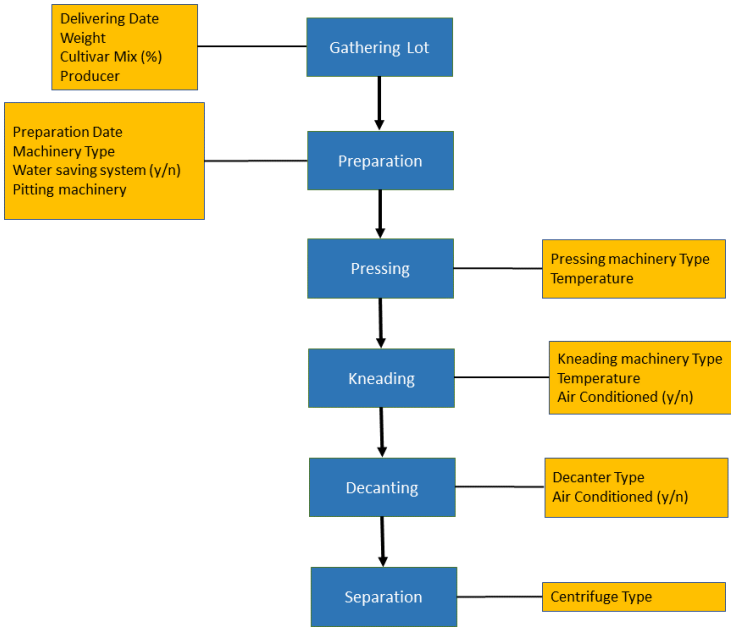


Figure 8.3: A snippet of PDO Olive oil Supply chain (II): the Crushing phase

that was produced during the crushing is determined to then favor the formation of larger diameter drops. This will make it easier to separate the oily phase from the aqueous one. In this phase, the temperature is an important parameter to control, as the paste should reach a maximum of around $25\circ C$. In this phase there are several procedures are designed to reduce the amount of water in the olive kneaded paste. From a practical point of view, three three different components: oil, vegetation water, and solid parts. Many variables are involved in this procedure: the amount of non-constitutive water introduced in the preparation phase, the temperature and the times of execution. There are three types of extraction technology generally used for this procedure: physical pressure, percolation and centrifugation. In physical pressure type a physical pressure guarantee the expulsion of the liquid after the milling phase: the paste is distributed on synthetic fiber filter discs, interspersed with metal discs. The set of these discs is placed on a trolley, equipped with a central metal guide column with holes. Pressure is applied to this "tower", using oil circuit hydraulic presses. In this way, the vegetation water is expelled together with the oil and both flow into the base of the trolley. The pressure is applied for about an hour. High quantities of stone fragments and low moisture content of the paste favor the extraction of the oil. Hence, this liquid is conveyed to the vertical centrifugal separator for the separation of oil from water, impurities and mucilage's. In the percolation type, a steel sheet is used which, immersed in a water-oil dispersion, remains

covered by an oily layer, due to the density of the oil (lower than water). This system allows to save only a part of the extractable oil (60-70%), so it must be repeated with higher pressure of centrifugal separator to allow a better recovery of oil from the pastes. The centrifugation type is widely used, as it allows to overcome many of the disadvantages associated with extraction by pressure. In this process, the oil paste is centrifuged in a rotating cylindrical conical drum with a horizontal axis (decanter). This process requires the addition of a small dose of water (around 10-20% of the weight of the olives). Due to the different specific weight, centrifugation operation separates two or three extracted solutions. Each of these solutions is separated by a decanters. There are three different type of decanters: the *three-phase decanter* separates the pasta into three fractions: oil pomace, oil must, and vegetable water; the *two-phase decanter* allows to use less water, and separates the paste into two fractions: oil pomace, and vegetable water; *the two-and-a-half-phase decanter* similar to the previous one, with another process that requires the addition of small quantities of water. The first method is based on the non-miscibility of oil and water: resting, the oil tends to rise to the surface, separating from the water. As soon as the pressing is done, a first oil decantation of the oil must allows to obtain a higher quality product. However, the amount of residual water requires longer times, and the storage of the oil in the oil mill in special masonry tanks. the second method is based on vertical centrifugation. This system consists of a cylindrical tank containing a rotating drum consisting of a series of perforated and superimposed conical discs. The oil must, introduced from above, enters the drum and is subjected to centrifugation at 6000-6500 rpm. Due to the different density, oil and water separate into two different outflows. During rotation there is an accumulation of solid residues (sludge) which are expelled. The product, in this phase, has some solid residues in suspension. Resting, these residues settle on the bottom of the container, and the oil becomes clear. For this reason, the freshly separated oil is stored in steel vessels, with nitrogen to avoid oxidation, in order to clean itself spontaneously.

The Oil Gathering phase

The oil obtained from the extraction often contains a minimum percentage of water, which is separated by oil decantation or oil centrifugation. The oil can stand for more or less short periods before being bottled. Storage can take place in the processing company or in silos at the bottling company, or in places used for the collection and storage of bulk oils. Important parameters of this phase are: avoiding oxygen, using closed containers or inert gases such as nitrogen; the temperature of the storage rooms; exposure to light; the presence of metals; contact with sludge. The type of storage tank and the temperature are key points to ensure the correct conservation of the

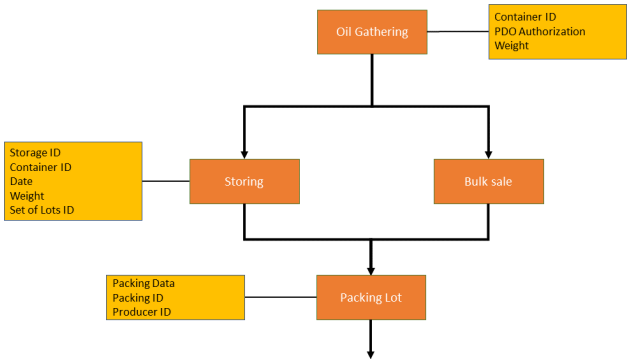


Figure 8.4: a snippet of PDO Olive oil Supply chain (III): the Oil Gathering phase

product. The factor that can most damage the quality of the oil is oxygen, which must therefore be managed with appropriate loading and unloading methods, which allow to minimize contact with the air. In the case of long-timed storage, it is need to use tanks equipped with valves and pipes for the introduction of inert gases for pressure regulations. The most used gas is nitrogen with less than 0.05% oxygen and without traces of halogenated and aromatic hydrocarbons. It is important to maintain the temperature between $10-25\circ C$ avoiding both overheating and freezing. An air conditioning system should therefore be provided if necessary.

Bottling phase

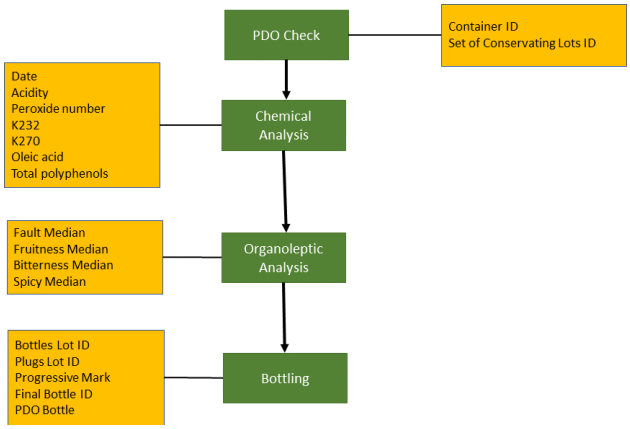


Figure 8.5: A snippet of PDO Olive oil Supply chain (IV): the Bottling phase

The oils must reach the consumer packaged in containers with a capacity not exceeding five liters (for catering, 25-liter packs are also allowed) with

hermetic closure and correct labeling. The bottling is performed by fully automated machines that ensure cleanliness and hygiene during the phases of the process. Before filling, the bottles are subjected to a jet of compressed air and overturning to facilitate the escape of any foreign bodies. After filling, it is possible to inject nitrogen into the bottle to replace the oxygen in the headspace. The containers must be inert to exclude the release of substances that could endanger human health, modify the composition of food products or deteriorate their organoleptic characteristics.

8.1.2 Representation of the PDO Olive Oil Supply Chain

To represent the process of the PDO Olive Oil supply chain, we have to adapt the phases described in Section 8.1.1 to produce a proper supply chain schema. From such schema we can draw the supply chain representation exploiting the DSGL of the *-chain framework.

The first step to obtain the graphical representation of a supply chain is to identify the relevant objects of the supply chain with respect to the focus of the analysis. Since the aim of our analysis is to trace the phases of the Olive Oil production process in order to accredit the origin of the goods, the main assets we are interested in are: olives, olive paste, dirty Oil and oil. The main properties of such assets, that are relevant for the production process are the following:

- The **Olive** asset represents a lot of olives, and it has the following properties: *PDO*, declaring the certification of the Olive asset; *producer ID*, indicating the ID of the Farmer who cultivated and harvested the Olive; *date*, indicating the date of harvesting; *quality*, defining the olive type (e.g., Lecchino, Arbequina, Calamata and others); *lot ID*, the id used to identify this Olive asset; *temperature*, representing the olive temperature kept during the storing; *Cultivar*, specifies which type of cultivation was used to grow the olives; *Preparation Date*, representing the date when the preparation phase was performed; *Machinery type*, representing the type of the engine used for the preparation: this property defines the tool used to remove dirty and unwanted elements in the Olive lot; *Whater System*, Boolean flag indicating the preparation type; *Pitting Machinery*, representing the type of preparation machinery used used to pit olives.

- The **Olive Paste** is the result of the "*Pressing*" operation performed in the Crushing phase (see Section 8.1.1), as shown in Figure 8.7. This asset represents the olive paste ready for decanting and, eventually, for the oil extraction in a centrifuge. This asset has the following

properties: *Temperature*, specifying the temperature of the Olive Paste at its creation; *Pressing type*, representing the type of pressing machinery; *Kneading type*, representing the type of kneading machinery; *Temperature*, representing the olive paste temperature during the kneading; *Air conditioned*, Boolean flag for the air conditioned presence; *Decanter type*, representing the type of decanter machinery.

- The **Dirty Oil** asset is the result of decanting process, from which oil can still be extracted for marketing. Although, the oil from this operation may not receive the PDO definition. Anyhow, this oil must still be traced in case it meets the quality parameters. This asset has only the *Centrifuge type* property, representing the type of the Centrifuge.

- The **Oil** asset has these properties: *PDO*, declaring if the Oil has the PDO definition; *weight*, representing the weight of the Oil after the crushing; *Storage ID*, the Oil container ID; *date*, the date of storing; *Set of Lots ID*; *BoL*, bill of loading paper in case of delivery; *Set of Conservation ID*; *acidity*, representing Oil acidity level; *Peroxide Number*, representing Oil peroxide level; *K232*, representing Oil specific component level; *K270*, representing Oil specific component level; *Total Polyphenols*, representing Oil polyphenols level; *Oleic Acid*, representing Oil oleic-acid level; *Fault Median*, a quality control parameter. *Fruitness Median*, a quality control parameter. *Bitterness Median*, a quality control parameter. *Spicy Median*, a quality control parameter. *Plug Lot ID*; *Progressive Mark*, the progressive number of the finish product; *Final Bottle ID*; *PDO Bottle*, declaring if the Bottle has the PDO definition.
  Hence, when this "*Oil*"asset is defined, the oil undergoes a series of chemical evaluations that define its properties. Once it passes these evaluations, the PDO definition is assigned, then the oil is ready for the bottling.

In the following, we describe how we represent the four phases of the schema shown in Figure 8.1 through our graphical language.

Preparation Phase

The Preparation Phase of the PDO Olive Oil production process is described in Section 8.1.1 and in Figure 8.2. The translation of this phase in the corresponding scGR is shown in Figure 8.6. The first asset – at the leftmost side of the schema– is the "*Olive*" one, which is an uncountable and consumable asset. This asset have an incoming arrow that is not originating from other

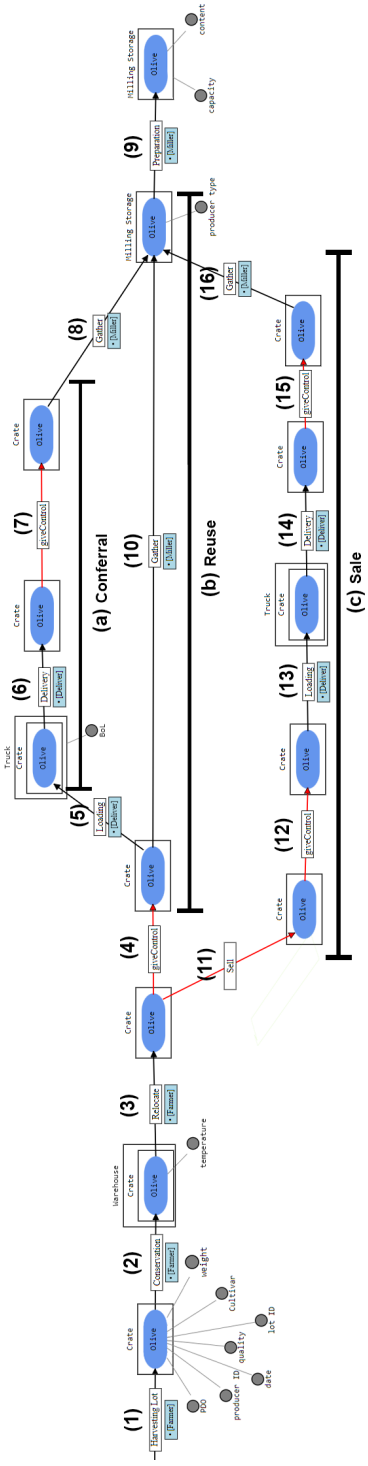


Figure 8.6: The Oline Oil Supply chain translation of the snippet in Figure 8.2, the **Preparation Phase**

assets, this means that this is the point of origin of any asset *Olive* because the production of this asset is not tracked using our SCM system. Hence, the asset *Olive* is generated using an *asset_ create()* operation, labelled as "Harvesting Lot" (labelled with "(1)" in Figure 8.6). The blue label "*Farmer*" under the operation name means that the creation of an *Olive* asset can only be performed by users holding the "*Farmer*" role. The asset *Olive* is also defined with a set of properties, which are: "lot ID", "producer ID", "date", "cultivar", "PDO" and "quality".

Since olives are an uncountable asset, to identify the lot a container is needed. The *Olive* asset is therefore contained within the "*Crate*" container. This container represents a crate, a basket or usually a large bowl, thus a generic container where the *Olive* asset is contained. After harvesting, the olives are moved for the storage step. Hence, the "*Conservation*" is the second operation that is performed on the *Olive* asset in our supply chain representation. The "*Conservation*" is an asset_pack() operation, which moves the crate object into the "*Warehouse*". the "*Warehouse*" is a non-consumable and countable container in which several crate objects may be placed and stored. This operation is labelled with "(2)". Meanwhile the Crate is into the Warehouse, the temperature of the oil is recorded. This value is represented by the "*Temperature*" property of the asset "*Olive*". After the resting time (around 24 hours), the olives are moved to the milling farm. Hence, the crate is extracted out of the storage warehouse: this procedure is represented by a asset_unpack() operation, labelled "*Relocate*". Since the "*Conservation*" and "*Relocate*" operations can be performed only by users holding the "*Farmer*" role, a blue label reporting such role is placed under the operation name in the scGR.

As described in the specifications of the **preparation** phase, the next step of the production process can follow three different paths, called: "Conferral", when the *Olive* asset is transferred for processing to another supply chain actor while maintaining the original owner, "Sale", when the *Olive* asset is sold to another user, and "Reuse", when the *Olive* asset is simply used in the next phase (see Figure 8.2). For the sake of comprehension, in the scGR of Figure 8.6 each of these paths is underlined with a black segment, labelled with "(a) Conferral" for conferral, "(b) Reuse" for reuse, and "(c) Sale" for sale.

The Conferral path and the Reuse path start with a "*giveControl*" operation, which transfers the control of the Olive asset either to a Deliver (for the Conferral phase) or to a Miller (for the Reuse phase).

The Conferral path is described as following. A "*Truck*" container represents the vehicle used for the delivering procedure. It is a non-consumable container with the bill of loading ("BoL") descriptor as the only property. The *Crate* containing the *Olive* asset is loaded into the *Truck* by performing

the "*Loading*" operation, and physically brought to the remote miller. The previous operation can be executed only by users holding the "Deliver" role. At the destination, the *Crate* containing the *Olive* asset is unloaded from the *Truck* and delivered to the miller, who become the new controller of the asset. These changes are represented by two operations: the "*Delivery*", which represents the asset being unloaded from the vehicle and the "*giveControl*", which represents the change of controller from the hauler to the miller. The user with the "Miller" role is the one that could load the asset *Olive* into the "*Milling Storage*" through the "*Gather*" operation. This storage is a non-consumable container, and is the starting point for the **processing** phase. Notice that the "*Gather*" operation is an asset_flow operation because it removes the *Olive* asset from the *Crate* container and stores it in the "*Milling Storage*", which is another container.

The Reuse path is actually very simple, because it consists of two operations only, a "*giveControl*" (label "(4)" in Figure 8.6), which represents the change of controller from the farmer to the miller, and the '*Gather*" (label "(10)" in Figure 8.6), with which the user with the "Miller" role loads the asset *Olive* into the "*Milling Storage*".

The Sale path starts with a "*sell*" operation (label "(11)" in Figure 8.6), establishing the change of owner of the *Olive* asset, that are sold by the farmer to another user. In our example, we suppose that all the involved users are following the same production process to obtain the certification, and hence they are interested in tracing it. For this reason, the tracing of the *Olive* asset continues also after they are sold. After the *sell* operation, in the same way as for the Conferral path, the controller of the asset *Olive* is changed with "*giveControl*" operation (label "(12)") because the asset is given to the hauler who will load it in his truck and will deliver it to the miller. Since the rest of the Sale path follows exactly the same procedure of the Conferral path, please refer to the above description.

Crushing phase

The Crushing Phase is described in Figure 8.3, and its translation to the corresponding scGR is shown in Figure 8.7.

When stored into the Milling Storage, the olive asset undergoes the preparation operation, represented by the arrow "*Preparation*" in Figure 8.7 (label "17") . This operation updates some of the Olive property such as: "Preparation date", "Machinery type", "Whater System", and "Pitting Machinery". After the preparation step, the Olive asset is ready for the first transformation process. We represent the "*Pressing*" operation with a dashed oriented black arrow, from the "*Olive*" to the "*Olive Paste*" (label "18" in Figure 8.7). The latter is an uncountable and consumable asset, still contained into the Milling Storage. This transformation operation defines two properties of

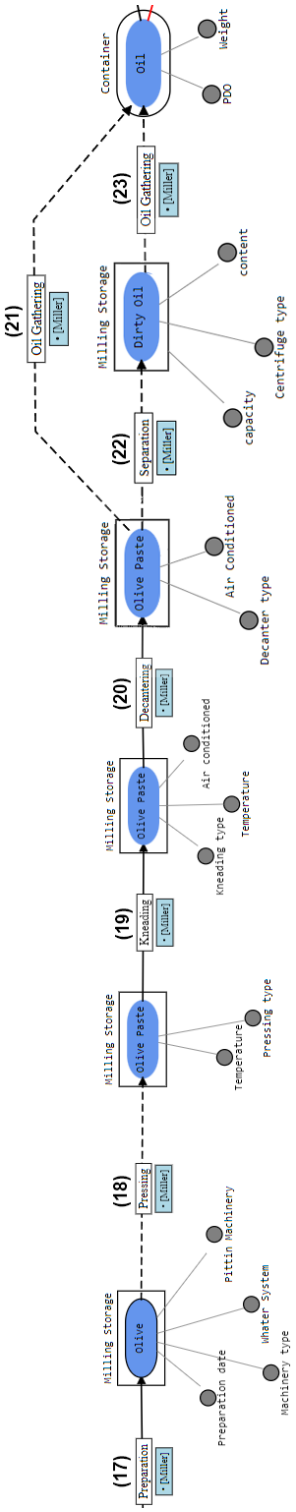


Figure 8.7: The Oline Oil Supply chain translation of the snippet in Figure 8.3, the **Crushing Phase**

the "*Olive Paste*" asset: "Pressing type", and "Temperature". Moreover, as shown in Figure 8.7, the *Pressing* operation can be executed only by supply chain participants holding the role "Miller". After the Pressing operation, the "*Olive Paste*" is kneaded. The the kind of kneading process, the temperature and the presence of Air Conditioning system impact the quality of the operation. Hence, the "Kneading type", "Temperature" and the "Air Conditioned" are properties of the "*Olive Paste*" that are updated as a consequence of the "*Kneading*" operation. This process is represented in the scGR by an update operation labelled as "*Kneading*". Finally, the "*Olive Paste*" is decanted in a long time process and its milling storage crate are stored into special air conditioned rooms. The descanting procedure leads to two different way of oil extraction: the Oil Gathering and the Separation. Separation is a further extraction procedure done on the olive paste, producing the "*Dirty Oil*" asset. Hence, "*Separation*" is a transform operation, linking the "*Olive Paste*" to the "*Dirty Oil*" (label "(22)" in Figure 8.7), which is is an uncountable asset stored into the "Milling Storage" crate itself. Since the "*Dirty Oil*" can be collected in various ways with the operation "Separation", so memory of the type of centrifuge is kept in the "*Centrifuge Type*" property. From the "*Dirty Oil*" asset, further "*Oil*" can be extracted through the "*Oil Gathering*" operation (label "(23)" in Figure 8.7).

Oil Gathering phase

The Oil Gathering phase is described in Figure 8.4. At this stage, oil is collected from both the olive paste and the dirty oil: all oils converge inside the silo. The oil at this stage of the process is generally stored for a certain period of time before bottling. Or, it may be sold to a other bottling facility than where it was produced. "*Oil Gathering*" operation is a transform operation linking the "*Olive Paste*" and the "*Dirty Oil*" to the "*Oil*" contained in its proper "*Container*". The "*Container*" is a consumable container provided by the "Miller". In the graphic representation, therefore, we define two different routes: one to represent the storing and the other dedicated to selling the produced oil. The arrow pointing to the upper side pair the storing operation, the *update* operation, paired with a black arrow, is labelled with "*Storing*" defines the following properties: "Storage ID", "Date", "Set of Lots ID" and the "weight". After a certain containment period, the oil is then moved into a containment tank together with other oils. Several oil assets may converge inside the Containing Tank, storing all the olive oil. In the lower route, we represent the sale of oil to a bottling facility. The "*sell*" operation defines the change of Owner from a user with the "Miller" role to a user with the "Bottler" role. The *sell* operation modifies the owner and the controller of the asset *Oil*. Hence, the *Oil* asset is loaded into a vehicle for the delivering. Last operation on this path is the "*Deliver*" to the Containing Tank.

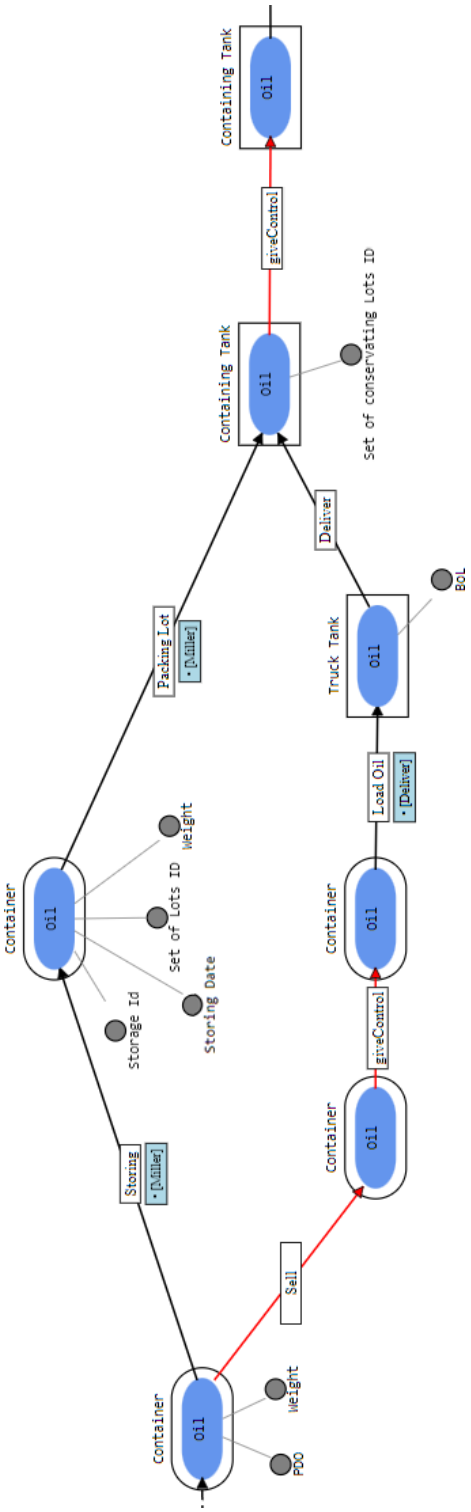


Figure 8.8: The Oline Oil Supply chain translation of the snippet in Figure 8.4, the **Oil Gathering phase**

Bottling phase

Before the bottling procedures for the next step, various PDO nomination checks must be performed: the oil asset has to be complies to these constraint. We represent these controls with the "*PDO Check*" operation on the asset Oil into the Containing Tank. The "*PDO Check*" could be performed only by a "Surveyor": a giveControl operation is then performed passing the control from the "Bottler" role to the "Surveyor" role.

The Bottling Phase is described in Figure 8.5. The Olive oil is gathered from the various resources and reuses containers: the Bottling phase starts. At this point the "*Oil*" asset is stored into a proper "*Containing Tank*". The "*Containing Tank*" is a non-consumable container provided by the "Surveyor". At this stage, the "Surveyor" executes two analysis operations to check oil important qualities: the "*Chemical Analysis*" and "*Organoleptic Analysis*"; the operation are performed sequentially on the "*Oil*" asset. The "*Chemical Analysis*" is an update operation, updating the properties: Data, Acidity, PEroxide Number, K232, K270, Oleid Acyd, and Total Polyphenols. The "*Organoleptic Analysis*" is an update operation, updating the properties: Fault Median, Fruitness MEdia, Bitterness MEdia, and Spicy Median. A final stage is bottling. Once the tests on "*Oil*" have been executed, the control of this asset is given to the bottler. Hence, the next operation is a "giveControl()' from the "Surveyor" to the "Bottler, performed on the "*Oil*" asset. The bottler performs a flow operation from the "*Containing Tank*" container to the "*Bottle*" container. The "*Bottle*" is a consumable container provided by the "Surveyor". From this point on, the set composed by the "*Bottle*" and the "*Oil*" asset is considered as the *Final Product*, ready for sale.

8.1.3 PDO Olive Oil SCMS Generated Interfaces

As described in Section 7.1, the *-chain framework generates three web interfaces starting from the scGR, meant to allow the supply chain administrator, the supply chain participants and the final customers to interact with the auto-generated smart contracts. In this section we describe the interfaces generated by the *-chain framework for the PDO Olive Oli SMS system.

Admin interface

The List of Roles section shows the list of roles in the Olive Oil PDO SCM system: Farmer, Deliver, Miller, Bottler and Surveyor. Instead in the Participant Registration panel, the manager grants to four Ethereum addresses the account as users. In this example, four users are declared: User1, User2, User3, and User4. Each of these users is associated with an Ethereum wallet

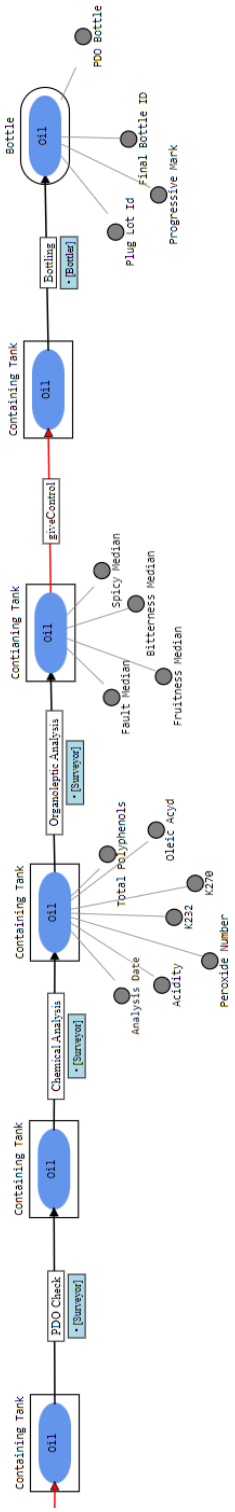


Figure 8.9: The Oline Oil Supply chain translation of the snippet in Figure 8.5, the **Bottling Phase**

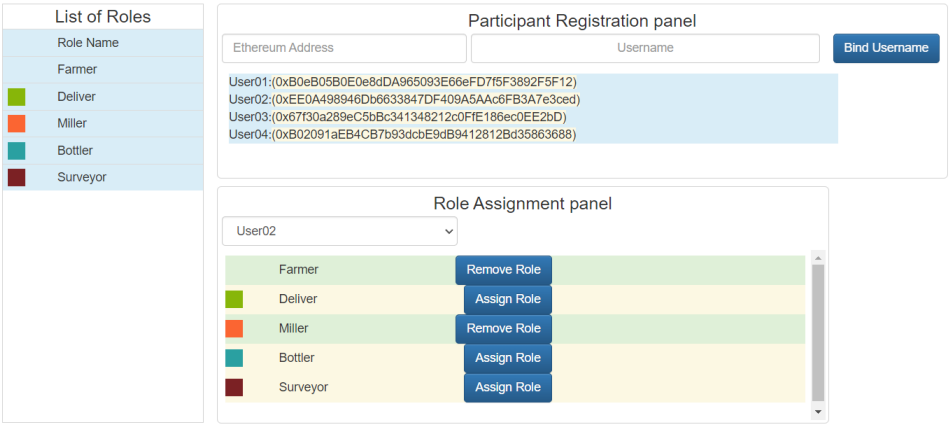


Figure 8.10: The admin interface panel

address. In the Role Assignment panel, the manager assigns to the User02 participant its roles. The *User2* is selected in the selection field. The list of roles of the selected user is shown in the bottom panel. The list of roles shows the User2 with two roles associated: Farmer and Miller.

Participant interface

Figure 8.11 shows a snippet of the graphical interface in which the Olive Oil PDO chain diagram is at a specific moment: it is represented as the olive asset contained in a crate that has just undergone the storage operation: here the participant may perform both a sell operation or a giveControl. In the second section, there are an input field and a list panel. The input field allow the participant to enter the name or an identifier of the product they want to create. Meanwhile, in the list panel, there is a list of all the assets the participant has right to manage. In this list, the item "OilLoot03" is selected among the others. This selection is highlighted in dark green. Below the supply chain diagram, the list of operations is shown. The operations are divided according to the assets. Each item in the list is represented with a transparent label. If the state in which the asset is located has performed the operation, its associated label is made dark grey and transparency is removed. All the possible operations following the state which the asset is located are displayed as "active". In this case, there are only three possible available operations: sell(), giveControl() and giveControl().

Viewer interface

The Figure 7.3 shows a snippet of the graphical interface in which the olive oil PDO chain diagram is at a specific moment: it is represented as the olive

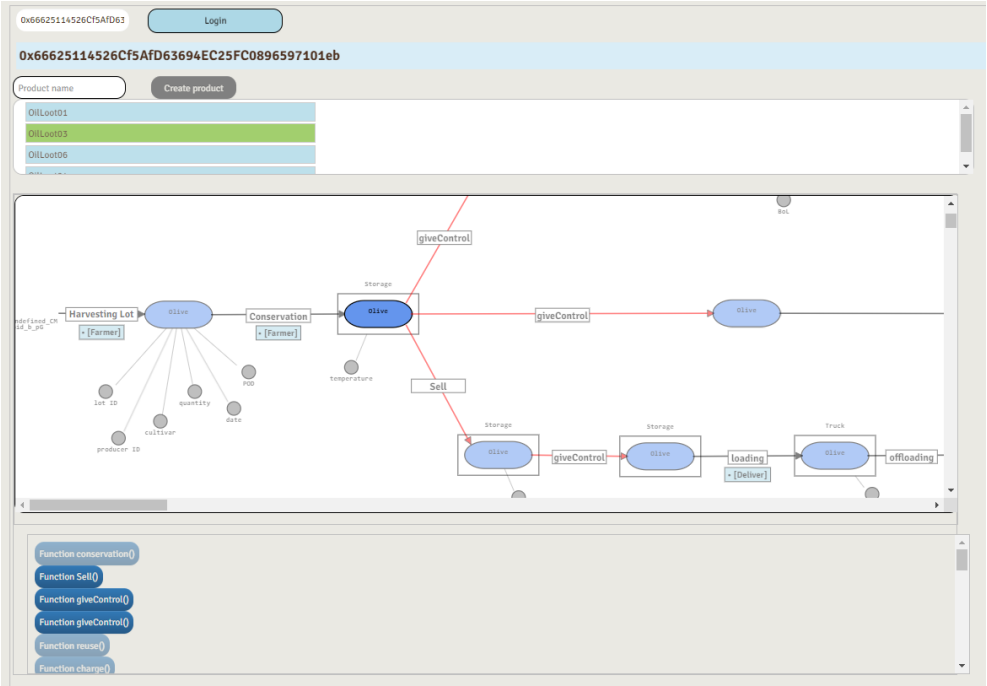


Figure 8.11: The participant interface panel

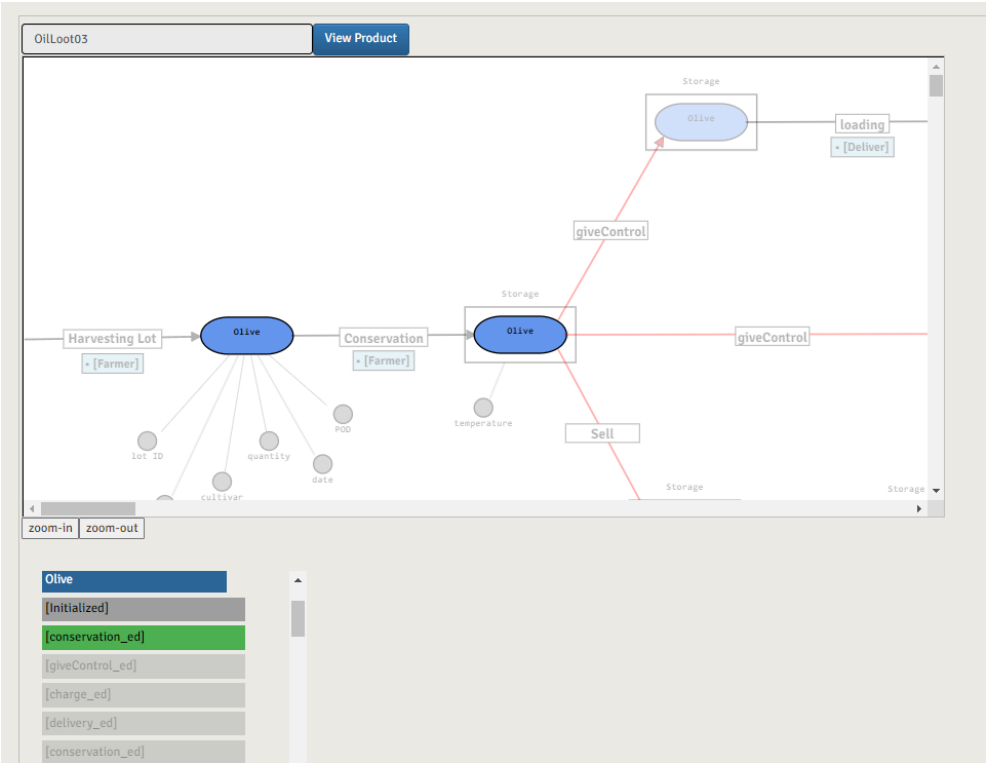


Figure 8.12: The viewer interface panel

asset contained in a crate that has just undergone the storage operation. In this case, the the user has selected the *OilLoot3* asset in the store of the smart contract. In the viewer panel all the information of this asset are displayed. All the assets of the scGR are show with an high degree of opacity. Instead the instance of the *Olive* asset of the *OilLoot03* In this specific case, the list of function already done shows the last function, which is *Conservation* with the corresponding state "conservation_ed", drawn in dark green in the interface.

8.1.4 PDO Olive Oil SCMS smart contracts costs

Asset	Deployment Cost	USD
Bottle	2758880	8.66
Container	2758908	8.66
Containing_Tank	2758894	8.66
Milling_Storage	2758894	8.66
Oil	11893905	37.35
Olive	6276689	19.71
Olive_Paste	5189055	16.29
Storage	2822497	8.66
Truck	2822497	8.66
erc20	1246301	3.75
Total	41286520	129.64

Table 8.1: Deployment costs of Olive Oil PDO SCMS smart contracts

This section describes in detail the costs of the PDO Olive Oil SCMS. In particular, we provide both the cost for setting up and to run the SCM system, via the Ethereum gas cost need for deploying the smart contracts, and for executing a full assets' life circle. Costs are evaluated via the execution of smart contracts through the Remix platform[2]. Table 8.1 shows the deploying cost of Olive Oil PDO SCMS as the sum of the costs of deployng all the smart contracts building up the SCMS. The cost are expressed in GWEI unit. Wei is the smallest denomination of ether, the cryptocurrency token on the Ethereum network. At the time of writing, 1 GWEI is equal to 0.00000314 USD, using the current live conversion rate. Technically, GWEI is a denomination of ETH – each GWEI is equal to 0.000000001 ETH. The average

[2] `https://remix.ethereum.org/`

transaction cost is roughly similar because the mechanics of each function within smart contracts are similar. The structure of smart contracts has a common base, consisting of management functions that set some fields of the main variables, plus transaction-specific functions. The operations functions represent the variable section of the automatically generated smart contracts. Thus, the cost of smart contracts depends on the number of operations on each asset and the properties involved on each function. The highest cost smart contract is the Oil asset contract, which has an amount of thirteen declared operations, for a total of sixteen total operations that also include the three standard *ERC20* initialization operations, asset creation, and generic object view.

To better characterize the example, we estimate the cost of the previously described olive oil production process, taking into account the assets of the process, controls, transformations and sales. To evaluate the total cost of tracing the production process, we take into consideration the worst case scenario, i.e., the one with the maximum number of operations in the scGR scheme. As we can see, the lowest cost is the "erc20" contract; in this case the cost refers to having to import the erc20 function interface as the smart contract extension that is automatically generated. Vice versa, the greatest cost is "Oil". In this case, the smart contract has many more properties and functions than the others, as well as having a smart contract builder who processes many properties for input. The fact of having many functions, implies that there are much more states in which the asset can be, thus more elements in the enumerable vector and consequent allocated operations.

Table 8.2, for each asset, shows the costs of the execution of each operation. Only operations of the following types were used: create, update, monitor, transform, sell (buy) and giveControl (takeControl). We can intuitively infer that the cost of the most of the operation is similar and relatively low. Although the assets represent very different real objects, many of them have a similar amount of ownership, but above all the same number of control constraints. These functions consist mainly in updating the state variable, resulting as the final state of the asset. Or, in the case of transformation operations, in calling an external function, i.e. the constructor of the new asset which the starting asset is transformed. The creation functions have -in general- the higher cost.

8.2 Use case: Carne PRI

8.2.1 Carne PRI Supply Chain Model

In the following, we describe the process that defines the production of the "Pezzata Rossa Italiana" (PRI) meat certification according to the Carne

Operation	Execution (gas)	USD
Bottle_create	24889	0,0809
Container_create	24967	0,0812
Oil_create	24967	0,0812
Oil_Bottling	22672	0,0737
giveControl	22867	0,0743
sell	22897	0,0796
Oil_Organoleptic_Analysis	22962	0,0796
Oil_Storing	22967	0,0796
Oil_Load_Oil	22964	0,0796
Oil_Packing_Lot	22934	0,0796
Oil_PDO_Check	22934	0,0796
Oil_Chemical_Analysis	22894	0,0796
Olive_create	24967	0,0812
Olive_Bottling	22672	0,0737
giveControl	22957	0,0743
sell	24967	0,0812
Olive_Charge	22861	0,0743
Olive_Conservation	22967	0,0796
Olive_Delivery	22867	0,0743
Olive_Pressing	22767	0,0740
Olive_Reuse	22680	0,0737
Olive_Paste_create	24967	0,0812
giveControl	22894	0,0796
sell	22934	0,0796
Olive_Paste_Kneading	22964	0,0796
Olive_Paste_Extraction	22938	0,0796
Olive_Paste_Separation	22934	0,0796
Olive_Paste_Oil_Gathering	22967	0,0796
Storage_create	24967	0,0812
Milling_Storage_create	24967	0,0812
Truck_create	24967	0,0812

Table 8.2: Execution costs of each function in the Olive Oil PDO SCMS smart contracts

PRI denomination. The phases of such production process are illustrated in Figure 8.13: each block of the flowchart identifies a phase, while the colours identify the actors performing that phase. Optional phases are represented with a dashed border. First of all, the *farmer* must identify the cattle by assigning a unique ID (*ID Assignment*), applied as a label to the animal's ear. The animal can then be either introduced into the herd (*Animal Introduction*) or purchased (Animal Purchasing). Both these options bring to the breeding phase (*Breeding*). During breeding in the barn, the *veterinarian* may administer antibiotics (*Antibiotics*). At this stage, there may also be an initial awarding of the *ANAPRI* Carne PRI Certification (*ANAPRI Certification*). After a four-month rearing period, the cow is ready for slaughter. Before transport, the specialised *veterinarian* evaluates the animal (*Cow Evaluation*). Then, the cow is transported to the slaughterhouse by one of the accredited *deliver* (*Cow Transport*). After the delivery, the cow undergoes a first analysis (*Ante Mortem Inspection*). Then, after the approval of the *veterinarian*, the *butcher* proceeds to slaughter (*Slaughter*). The carcass must rest for 48 hours before being processed, passing the second evaluation (*Post Mortem Inspection*). Then, the technician gives the carcass stamps and codes delivered to ANAPRI and the ID Mark of the animal (*Carcass Evaluation*). If the butcher's shop is not equipped to handle the carcass properly, the meat is left in the blast chiller (*Carcass Maturing*) and is only then transported to a specialised butcher's shop (*Carcass Transport*). In the specialised butcher's shop, the butcher cuts up the carcass, dividing it into homogeneous parts for processing (*Sectioning*). If the meat has not been placed in the blast chiller, it is left on rest for about 48 hours (*Meat Maturation*). The chopped meat is then transported to the sale point (*Meat Transport*), where other butchers will either grinds (*Grinding*) or cut (*Portioning*) the pieces of meat. Each preparation is then packaged (*Packaging*) to be ready for sale (Selling).

8.2.2 Representation of the Carne PRI Supply Chain

Each block in the diagram of Figure 8.13 is translated into a set of operations of our DSGL.

The phases ID Assignment, Animal Introduction, Animal Purchasing, Antibiotics and Breeding are shown in Figure 8.14. The pattern begins with the *Cow_ create* operation (1); it is a create operation which only the Farmer can invoke. In this case, the operation lead to countable and consumable asset: Cow. The *ID mark* property is described as the only initial property of the Cow asset. The Animal Introduction is paired by a bifurcation in the graphical representation. In the first case, the cow asset is purchased from an external Farmer, then the buy and sell is represented. The *Sell* operation (2) -with the Buy implication- and a *giveControl* operation (3)-with the

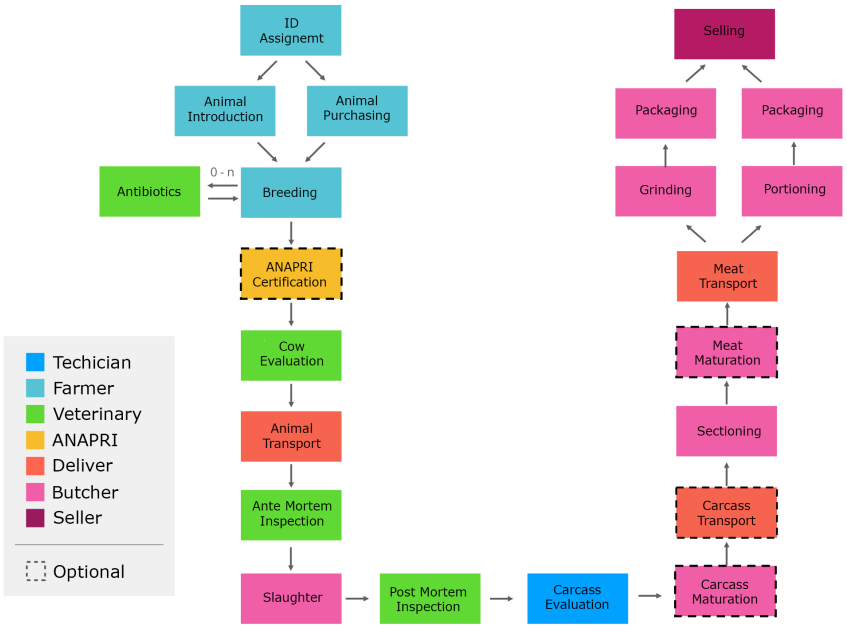


Figure 8.13: The Carne PRI schema.

takeControl implication- on the Cow asset is performed. At this stage, it was not necessary to keep track of the movements by other means of the Cow; the relevant part was the definition of the breeding phase. The *Cow_ breeding* function (4) that can only be used by the Farmer, which determines the cow's four months of breeding, is then invoked. It is then represented by modifying the *4 months start* property of the Cow asset. In the second case, the specified Farmer is the owner of the cow. The *Cow_ breeding* function (5) is then simply invoked, which has the same behavior as the previous one described. The Animal Purchasing phases is paired with a bifurcation in the graphical representation, as an optional and parallel activity. In the first case, a Veterinarian must take charge of the administration of Antibiotics. A necessary *giveControl* (6) is then first submitted in which the Veterinarian can have access to operations on the Cow asset. Since this is a repeatable operation, the monitor operation is introduced: the *Cow_ antibiotics* operation (7) is executed the first time. Within the *Cow_ antibiotics* operation (8) the constraint of the maximum number-and optional minimum number-for which the operation can be executed is specified. When finished, a new *giveControl* (9) is executed in which the Farmer regains control of the Cow asset. The *Cow_ endOfBreeding* (10) update operation represents the final state of the Cow asset at the end of the four months of breeding, updating the *4 month*

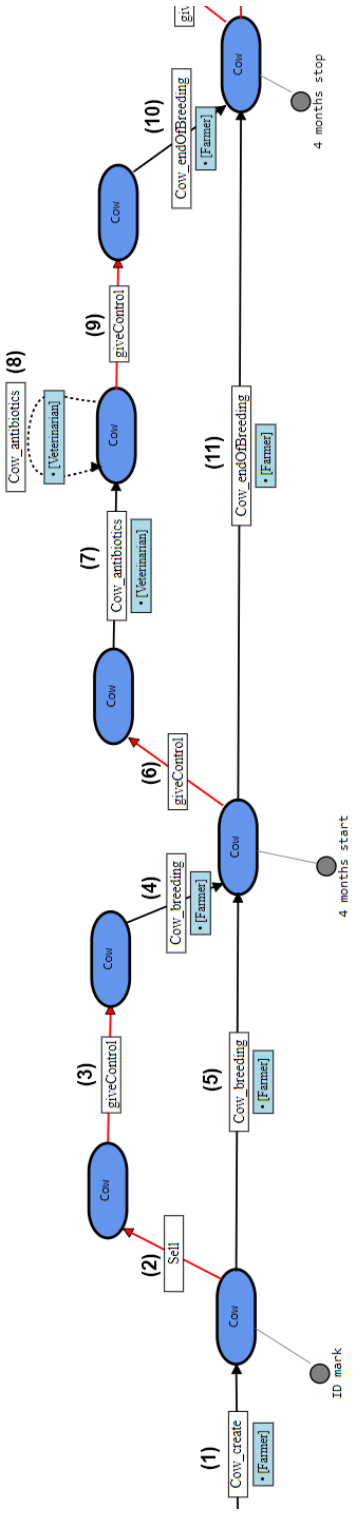


Figure 8.14: Snippet of Carne PRI Supply chain translation of the Figure 8.13 schema representing the phases ID Assignment, Animal Introduction, Animal Purchasing, Antibiotic Administration and Breeding.

stop property of the Cow asset. In the second case, the veterinarian does not administer any antibiotics, so only the emphCow_endOfBreeding (11) function is performed, as the previous one described.

The ANAPRI Certification, Cow Evaluation and Cow Transport phases are shown in Figure 8.15. This scheme begins with the optional Anapri certification. This choice is then represented with a bifurcation in the diagram. In the first case, a *giveControl* (12) is performed in favor of the ANAPRI operator on the Cow asset. Then an update operation *Cow_ ANAPRICertify* (13) is executed that establishes ANAPRI certification. The Cow asset undergoes a *Cow_ evaluation* (14) update operation that updates the transport suitability property, thus ensuring that it can undergo transport. In the second case, the veterinarian does not administer any antibiotics, so the emphCow_evaluation (15) functions is performed. The next step, is to load the cow for transport. A *giveControl* (16) is performed to allow the Deliver the ability to perform the *Cow_ load* operation (17) inside the non consumable container *Truck*.

The Ante Mortem Inspection and the Slaughter phases are shown in Figure 8.16. The Deliver delivers the cow with a *Cow_ deliver* (18) update operation, allowing the Veterinarian to perform the antemortem checks. The *Cow_ anteMortemIspection* (20) operation is an update operation that updates the standard organolectic property. Next, the cow is subjected to slaughter, then to the Cow asset transformation. A *giveControl* (21) is performed to allow the operation to be executed at the Butcher. The Butcher executes the *Cow_ Slaughter* (22) operation by creating the Carcass asset with the *ANAPRI ID* propery.

The Phases of Post Mortem Evaluation and Carcass Evaluation are shown in Figure 8.17. Next, the Carcass must undergo quality control, permissions are then requested for the Veterinarian via a *giveControl* (23) to allow the *Carcass_ postMortemInspection* update operation (24). Finally, if all the steps have been correct, an ANAPRI technician must non-optionally assign the ANAPRI code and ANAPRI labeling on the Carcass in this case. Through the *Carcass_ evaluation* update operation (26), the Technician updates the *ANAPRI standard* property.

The Carcass Maturing and Carcass Transport phases are shown in Figure 8.18 This procedure is also represented by a bifurcation. In the case where the Carcass is not to be transported to another implant, the Carcass undergo a culling period, represented by the *Carcass_ maturation* function (28) executed by the Butcher. The temperature and moisture properties of the carcass asset before transport are updated. Transport, involves a similar procedure as seen in the figure, where the operations described (29-32) represent the Deliver takeover, the transport of the carcass -*Carcass_ load* (30) and *Carcass_ deliver* (31)- until the delivery to the Butcher with a proper

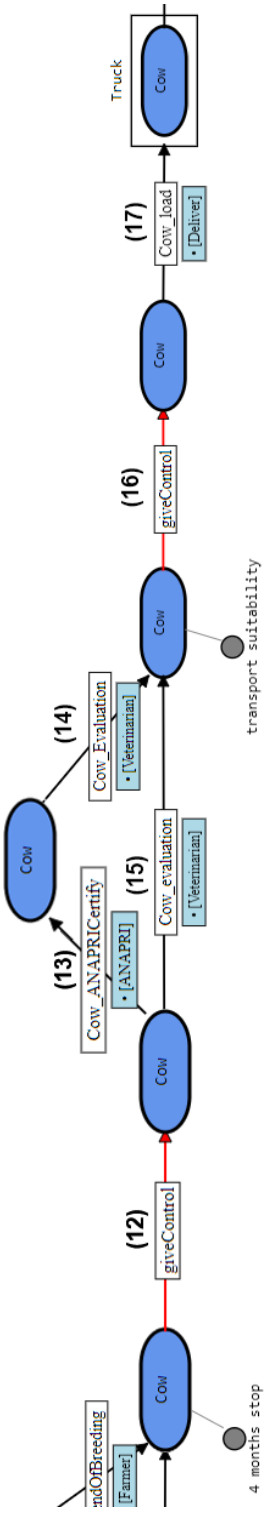


Figure 8.15: Snippet of Carne PRI Supply chain translation of the Figure 8.13 schema representing the phases ANAPRI Certification, Cow Evaluation and Cow Transport.

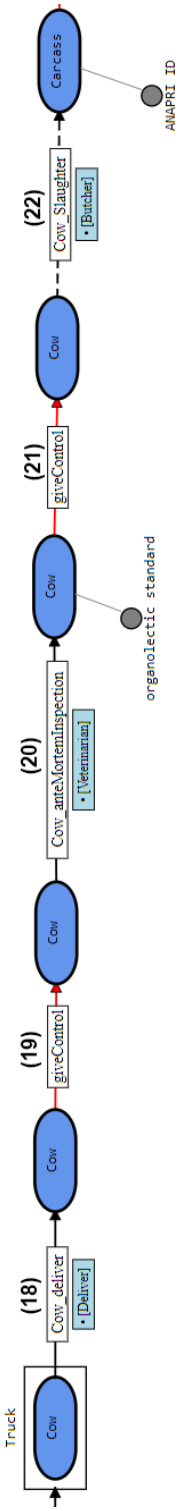


Figure 8.16: Snippet of Carne PRI Supply chain translation of the Figure 8.13 schema representing the phases Ante Mortem Inspection and Slaughter.

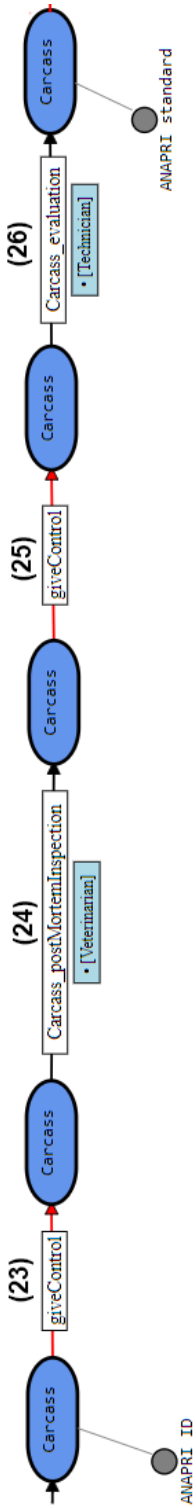


Figure 8.17: Snippet of Carne PRI Supply chain translation of the Figure 8.13 schema representing the phases Post Mortem Evaluation and Carcass Evaluation.

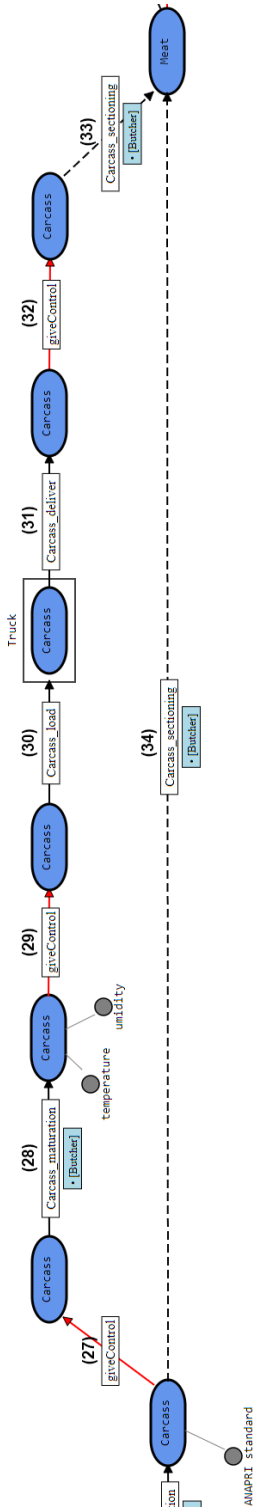


Figure 8.18: Snippet of Carne PRI Supply chain translation of the Figure 8.13 schema representing the optional phases Carcass Maturing and Carcass Transport.

giveControl (32) operation. Once the Butcher has control of the Carcass, a further transformation operation is performed, which destroys the carcass by splitting the meat pieces. Then, The *Carcass_ sectioning* (34) operation generates the Meat asset.

Meat maturation, and Meat Transport phases are shown in Figure 8.19. Depending on the piece of meat and its destination, there is a culling step, to maintain the properties of the meat. Then an optional bifurcation represents the ripening stage. The *Meat_ maturation* update operation (35) specifies the chilling status of the meat. The meat thus produced then undergoes transport to the point of sale. The Deliver is instructed to deliver the meat via a *giveControl* operation (36-37) in order to load the Meat asset into the non-consumable Truck container with the *Meat_ load* (38) asset_pack operation.

The Grinding, Portioning, Packaging and Selling phases are shown in Figure 8.19 Once the Meat has been delivered by the Deliver via a *giveControl* (40), the Butcher at the point of sale can arrange to sort the various formats of the Meat. One part will be submitted to the grinder, represented by the *Meat_ grinding* transform operation (41). This operation generates an uncountable Mincemeat asset, packaged in the consumable container *package*. In another process, the Meat is cut and divided into slices. The *Meat_ portioning* (42) operation is performed to represent the cut, i.e. a transformation operation that generates the Steak asset. This operation generates an uncountable Steak asset, packaged in the consumable container *package*. Both products are now ready for sale, which is not represented in the current scheme, nor in the SCM system logic.

8.2.3 Carne PRI SCMS smart contract costs

We present an analysis, describing in detail the costs of translation of the model into an automated SCMS. We want to provide quantitative proof of the costs for both the management and the use case, via the ethereum gas cost need for deploying the smart contracts, and for executing a full assets' life circle on the SCMS. Costs are evaluated via the execution of smart contracts through the Remix platform [3]. In the Table 8.3 we show the deploying cost of CarnePRI SCMS. The cost are expressed in GEWI unit. GWei is the smallest denomination of ether, the cryptocurrency token on the Ethereum network. Currently, 1 GWEI is equal to 0.00000314 USD. Using the current live conversion rate. Hence, GWEI is a denomination of ETH – each GWEI is equal to 0.000000001 ETH.

To better characterize the example, we try to estimate the total cost -with respect to the blockchain- of the product's life path, through the various as-

[3] `https://remix.ethereum.org/`

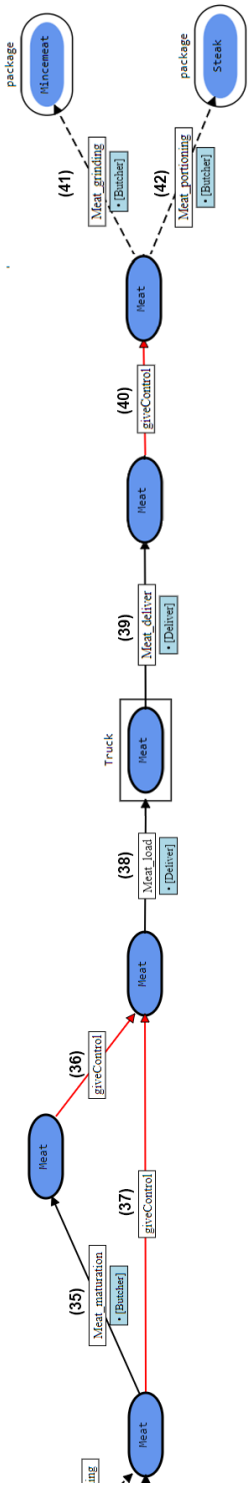


Figure 8.19: Snippet of Carne PRI Supply chain translation of Figure 8.13 schema representing the phases Sectioning, Meat Maturation and Meat Transport.

Asset	Deployment Cost	USD
Cow	6026082	18.13850682
Carcass	4959541	14.92821841
Meat	3710175	11.16762675
Mincemeat	2609557	7.85476657
Steak	2609407	7.85431507
Truck	2618557	7.88185657
Package	2609544	7.85472744
erc20	1246301	3.75136601
Total	26389314	79.43183514

Table 8.3: Deployment costs of CarnePRI SCMS smart contracts

sets, controls, transformations and sales. We calculate at least one execution for each step, taking the longest path in the paths the SC provides. The longest life path of a single product has a maximum of 90784 (GWEI) gas cost, equivalent to 0.29 USD.

The Table 8.4 shows the costs of the execution of each individual operation: all functions declared in smart contracts are listed in the table, divided by asset type: *cow*, *carcass*, and *meat*. Only the asset cow has an explicitly stated *create* function, since every other asset originates from a new transformation. The average cost of an operation is around 22k units of gas. As we have described, all types of DSGL operations were not required to drawn the scGR. Only operations of the following types were used: create, update, monitor, transform, sell (and buy) and giveControl (and takeControl). We can intuitively infer that the transformation operations have a relatively lower cost than the update and monitor operations, since that these functions consist mainly in updating the state variable -resulting as the final state of the asset- and invoking the constructor of the asset class into which the starting asset is transformed. The creation functions have -in general- the higher cost, because they are setting value to all the initializing properties. In the asset *cow* these function are declared: give_control, cow_breeding, give_control, cow_antibiotics, cow_endofbreeding, carnepri_certify, cow_evaluation, cow_load, cow_deliver, cow_antemorteminspection, cow_slaughter In the asset *carcass* these function are declared: carcass_create, carcass_postmorteminspection, give_control, carcass_maturation, carcass_load, carcass_deliver, and carcass_sectioning. In the asset *meat*

Operation	Execution cost (gas)	USD
cow_create	22791	0,0741
sell	22657	0,06366617
give_control	22635	0,06360435
cow_breeding	22678	0,06372518
cow_antibiotics	22637	0,06360997
cow_endofbreeding	22680	0,0637308
carnepri_certify	22700	0,06378700
cow_evaluation	22658	0,06366898
cow_load	22612	0,06353972
cow_deliver	22672	0,06370832
cow_antemorteminspection	22682	0,06373642
cow_slaughter	22492	0,06320252
carcass_create	22791	0,0741
carcass_postmorteminspection	22682	0,06373642
give_control	22635	0,06360435
carcass_evaluation	22635	0,06365212
carcass_maturation	2635	0,06359592
carcass_load	22608	0,06352848
carcass_deliver	22672	0,06370832
carcass_sectioning	22492	0,06320252
meat_create	22791	0,0741
meat_maturation	22667	0,06369427
give_control	22635	0,06360435
meat_load	22608	0,06352848
meat_deliver	22672	0,06370832
meat_grinding	22492	0,06320252
meat_portioning	22492	0,06320533
mincemeat_create	22791	0,0741
steak_create	22791	0,07411

Table 8.4: Execution costs of each function in the CarnePRI SCMS smart contracts

these function are declared: meat_create, meat_maturation, give_control, meat_load, meat_deliver, meat_grinding, and meat_portioning. In the asset *mincemeat* only mincemeat_create is declared. In the asset *steak* only steak_create is declared. In addition, three functions common to all auto-generated assets are implemented: asset_view, asset_destroy, and the ERC20 label contructor.

# Cows	Total Execution cost (gas)	Total cost (gas): Deployment + Execution	USD
1	90784	28470712	89.40
2	181568	28561496	89.69
3	272352	28652280	89.97
5	453920	28833848	90.54
10	907840	29287768	91.97

Table 8.5: Total execution costs of the CarnePRI SCMS.

The Table 8.5 shows the total costs of a hypothetical product life cycle. The Table 8.4 is considered, selecting from it the "longest" asset path in the scGR. Then the costs are summed, multiplying -where necessary- according to the cuts made on an adult cattle. Two halves can in fact be obtained from a bovine, multiplying the costs of operations (from that point on) by two. The Table 8.5 was calculated exploiting a particular use case in which a cow is initially given a course of five antibiotics. The steps involved in this specific supply chain use case also consider all the steps of delivering by truck and change of the control of the product between different (roles) accredited operators.

CHAPTER 9

RELATED WORKS

9.1 Related Works

In this thesis we present the results of our previous works [4, 3]. In particular, in [4] we presented the initial idea of the supply chain representation model and of the DSGL, which are then exploited in [3] to represent a small excerpt of a well known supply chain. We refined the proposed model and DSGL by adding some new operations in order to allow a better representation and differentiation of the change of owner (*asset_sell/asset_buy* operations) and of controller (*asset_giveControl/asset_takeControl* operations) of an asset. We also refined the representation of the authorization rules that are paired with operations to regulate their execution by supply chain participants. We also provide the detailed description of the complete workflow of the creation of SCM systems (see Figure 3.1), including a detailed description of all the components of the *-chain framework and the interactions among them. Finally, we give a complete representation of the well known supply chain we introduced in [3] as reference example: in particular, we represented all the assets and all the operations of such reference example through the refined version of our DSGL, including the changes of owner and of controller of the assets. Moreover, the refined version of our DSGL also allowed us to represent a proper set of authorization rules and constraints paired with the operations of such supply chain, while these concepts had been just mentioned in our previous works.

First of all, we evaluate the various types of SC implementation and the different applications described in the literature: we analyse the state of art of main SC implementation, then comparing the main solutions with the ones described in other works. We analyzed the main SC literature models and

how they are implemented, highlighting for in the various studies addressed the main problems: difficulty of representation, difficulty of design, difficulty of automation.

An example of the power of representative modelling for SC is described in [54]: here the authors develop a simulation model, using an object-oriented modelling framework to facilitate supply chain representation. We would like to highlight that this approach -like many others- is strictly process-centric, where the state or the maintenance of each asset is often not considered. In this theoretical framework they point to solve one of the main problem of SC: to easy represent the SC schemes. The object-oriented approach ease to represent the components of a SC. However, the study refers only to the processes and not to the assets involved. In this way, each phase is represented by an object with the main "execution" function. Other timing and control operations are then implemented. The framework does not address who performs the operations of the SC.

An automated approach [38], deals with the translation of UML into other languages. The study presents a tool capable of semi-automatically translating a specific structured UML schema into an activity diagram (AD). The AD is then subjected to an analysis using the mCRL2 framework capable of establishing both to earn an evaluation of constraints over the model and to grant further translations in structures compatible with the XML language. The study aims to provide to the UML schema a greater power of representation, and allow to translate it -under certain structural and protocol constraints- into a generic format, useful for other processes. However, the project is limited to the use of java libraries and the massive request for computation. There is also no actual automation of the entire process, as only two of the three main parts are automated: the fundamental step through the mCRL2 framework has not been implemented.

Another automated approach, [38], deals with the translation of UML into other languages. The study presents a tool capable of semi-automatically translating a specific structured UML schema into an activity diagram (AD). Hence, the mCRL2 framework analyses the AD: the framework evaluates the constraint of the model and translates it into an XML-compatible structure. However, the graphical representation makes the design and construction of smart contracts much easier, not only for the transactional purpose.

The study in [12] presents an integration of the BCautoSCF framework, the framework deals with the management of the financing platform for the auto retail industry, from the gathering of the components to the sale. The newly developed tool aims to automatically track the steps described by the BCautoSCF framework.

The Business Process Model and Notation [55] (BPMN) is a model language to design business processes. This model allows to define the workflow,

the participants, the choices of the process flow. The drawn schemes are designed to be detailed, but easy to read without training. A BPMN schema does not directly translate to any specific language or implementation. The BPMN schemes are process-oriented and often not consider or track the involved asset. Thanks to all of those features, the BPMN model is used in most of the common SCM System designation.

In [26] the authors show how the Supply Chain Operations Reference (SCOR) model, and the BPMN model can be combined: both are process-oriented workflows. The study shows the obvious similarities of the two models, which see the supply chain as a sequence of operations, focusing on the execution of being, without keeping track of what and where the objects involved in the operations are: in these models, there is no concept of 'assets', nor of asset ownership. In our framework, the concept of asset is fundamental: the objective is to keep track of the life cycle of the asset, who owns it, and who has had the opportunity to use and/or modify it.

Many of the solutions we have given space to have distributed technologies as a scenario. In these cases, we talk about systems designed around blockchain logic.

In [51], blockchain and IoT based solutions are proposed for agri-food supply chain tracing system. This study focuses on a specific use case representing an SC to trace products "from farm to table", and compares the results obtained implementing such system on Ethereum and Hyperledger.

Authors in [34] have introduced blockchain-based food information security in SCM System. According to them, no actual solutions can achieve the traceability accuracy required for the real market: although a solution based on distributed technology is provided, even if that an implementation has not been developed.

The study case presented in [50] shows a financial systems applied to blockchain technologies. In this work, they develop an economic exchanges tracking system in order to increase the reliability of the trustness trade. The implemented system represent a theoretical financial case.

The authors of [9] propose a supply chain solution for the wine industry based on blockchain technology. The solution exploits RFID tags and barcodes to identify the assets, and records the data relevant for each stage of the supply chain on the Multichain platform in order to ensure the quality of the production process. Despite it is close to ours, this proposal is specific for the wine supply chain, and it is not aimed at being general supporting the tracing of customized SC. A similar blockchain-based reputation system is proposed in the [60]. In this solution, an IoT network provides data credibility based on a voting system: each message between two nodes represent a transaction that is revised and validated by the nearest nodes in the network.

A similar approach is presented in [15], where common basic elements of

BPMN are used through the Choreography graphics engine: these predetermined elements of the graphics engine allow to represent various processes in simple diagrams. These schemes are then parsed into Chaincode smart contracts for Hyperledger. In this scenario, the presented framework focuses on tracing the process for the production of a good, instead to treace the good itself. Moreover, different actors participate to write the SC schema, introducing more complex levels of errors. Furthermore, the blockchain technology on which the generated smart contracts are adapted is Hyperledger Fabric, i.e. a private permissioned BC. In a different way, our tool is developed for a single domain expert, avoiding external influences. Also, the goal is to generate smart contracts for Ethereum systems, which are known not to represent private systems.

A more recent study [8] shows how in the complex problem scenario of the SCM System, traceability of an asset is preferable in respect to the costs (production and maintenance) of an SCM system itself, all through blockchain technologies that in each case reduce the costs of the SCM System.

In the BPMN choreography scenario in [43], a particular extension of BPMN is used to empower the description of processes with the participation of external parties or domain expert. The presented methodologies are developed to ensure automation in the creation of smart contracts for graphical models built with BPMN choreography tools. In this case, the methodology uses previously built smart contracts, on which additional extensions and functions are applied. Furthermore, the traceability problems of a product are not taken into account, only the execution steps of a process are traced by this method. In our framework, on the other hand, the possibility of having an own graphical model -free of fixed paradigms- allows us to develop a more adaptive translator using blocks of a own language that are easy to understand, but at the same time can represent complex semantics.

In [41], a model based on the Business Process Model and Notation (BPMN) representation is shown. The developed graphical DLS translates blockchain smart contracts using the graphical representation of the DEMO modelling language [17]. This representation makes it easier for the user to represent a workflow or the transaction operations of the same asset. Due to the nature of the BPMN representation, it is difficult to design various types of SC: this representation is hardened and the schemes that can be drawn are limited

Several works in literature analyse SCM systems, but most of them are focused on the management of a single use case or in the best scenario they analyse a specific SC topology. For example, the [37] shows an analysis of possible SCM system solution, presenting however theoretical studies. In literature, different studies analyse SCM systems, from the point of view of a single use case, or a particular asset: surveys like [28] analyze possible

development of SCM systems in different architectures introducing the idea to exploit DLT. The comparative study presented in [32] analyzes the limits of the usual BPMN and Case Management Model and Notation (CMMN) approaches in the representation, stigmatization and possible translation into smart contracts. The authors highlight how in BPMN only specific types of processes can be represented, limiting further abstractions or exceptions; while the representation in a more abstract model, the CMMN, is limited for what concerns its representation on the blockchain. The survey [33] presents a large collection of current agri-food supply chain application solutions. Most of these solutions rely on blockchain infrastructure. These solutions use a proper infrastructure integrating their codes in smart contracts for Hyperledger fabric.

In [22], discusses about the topic: "how to increase the competitiveness of a production". The main answer through distributed technologies is discussed. In the study, the use of a public permissionless blockchain is examined.

Another approach, [59], involves the development of a contract modelling language: this approach provides a high-level language –very similar to natural language– capable of representing some types of transactional operations. However, this approach makes it difficult for a user to design a supply chain: first of all, it requires learning the high-level language presented in the study. A similar study, presented in [30], describes a tool that builds smart contracts using precomposed functional blocks. The idea is that composing such blocks is easier than developing complex smart contracts. Unlike this approach, our solution does not include pre-generated blocks, allowing the users to define their low level or high level operations via the graphical interface.

The authors of [13] present an automatic smart contract generation framework exploiting ontology and semantic rules to represent the domain specific knowledge, from which smart contracts templates are derived. Abstract syntax trees are then exploited to organise the constraints derived for each specific setting, and are used to produce the final smart contracts. Similarly to our framework, this proposal aims at being very general, not focused on a specific problem or domain. The main difference is that our framework allows its users to design their SC through a user-friendly DSL implementing the general model we defined.

Another kind of solution is proposed in [40] where each transaction involves a registration on the blockchain and in the Interplanetary File Storage System (IPFS[1]), which is a secondary storage system. In particular, the data stored in the IPFS are paired with a hash, which is recorded in the blockchain.

One of the to this paper most similar study of [30] presents a tool capable to design solidity code based on predetermined logic blocks (easier to

[1]https://ipfs.io

understand than software source code). This solution is based on a virtual environment that allows to build a smart contract by giving easily understandable bricks. Unlike this approach, our solution does not include pre-set bricks or a few common combinations of solidity code: DSGL aims to be as general as possible such that it can represent multiple combinations of SC.

Instead of the classical product life-cycle model, [58] presents a process validation based on blockchain system: the idea is to create an accurate digital representation of the end product. A physical good is represented as a collection of cryptographic tokens, and different combinations of tokens describe a different physical good. The list of cryptographic tokens of each product describes the recipe of the product.

In [11], the authors present a comparative study and a practical realization of a solution for the food traceability problem. In this case they use both the RFID technology and an instant picture of each asset under exam. This pair of information is recorded through a single transaction on the blockchain at each step of the supply chain process.

[48] shows another application of RFID tags using "Hazard Analysis and Critical Control Points" (HACCP) for the tracing of the crop supply chain. The main idea is to pair each stock of harvested crops with an RFID tag, and then to follow the tag movements recording such information on BigchainDB. The temperature in the stocking area is monitored using IoT devices. Again, this work is focused on a specific supply chain and, differently from our framework, the design of customized smart contracts implementing specific tracing system is not supported.

[25] proposes an Ethereum-based network that works as a digital certificate of authenticity for 3D design intellectual property assets. They have integrated blockchain into OpenDXM GlobalX software, which is used by manufacturers for sharing data. Each licensor has a private key: a digital certificate of authenticity is created with this key. The certificate and the key are recorded in the blockchain.

Similarly to the previous approach, authors in [24] propose an efficient strategy for Agri-Food supply chain traceability, where goods provenance data (e.g., images, videos and data collected from sensor) are stored in the Interplanetary File System, and the blockchain is used to store the IPFS hash address of such provenance data.

[56] proposes a decentralized storage beside the ethereum blockchain. The study highlights the risks of centralized storage: sensitive data could be leaked, and the IPFS itself can lose information in the event of a direct attack on the system. To solve these problems, they propose to use a file encryption algorithm and to record the obtained hash in ethereum blockchain.

A case study on product traceability is presented in [27] where the authors describe "originchain". The framework use two blockchains: one is a

private blockchain for off-chain transaction recording. The hash of the off-chain recording is then saved in a public blockchain (on-chain transaction recording). The idea of mixed private-public blockchain is not at design level but can be decoupled also in our framework and applied also in the context of our automatic translations. It is enough to deploy additional smart contract that transfer on the public blockchain information saved on a private ones periodically.

In [2], they present two smart contracts aimed at regulating an agricultural supply chain integrated into a blockchain system. The supply chain so described is poor and keeps track only of the economic transactions between three actors: the manufacturer, the trade center and the final customer.

In this [21] about vaccines, it is shown how with the introduction of blockchain technologies the distribution and administration of high-impact vaccines to the population would bring more control and easy verification by the main authority. The study does not propose an actual supply chain schema, but only an hypothetical schema; however, a programming solutions to manage the data and its storage is proposed.

[52] proposes the Gcoin project[2] to record transactions between pharmacies and consumers. Here the transactions on the blockchain maintain information with the aim to identify drugs and possible illicit transactions. Only authorized users can sell/buy specific items through this platform, with the goal of avoiding both counterfeiters and unauthorized buyers.

[35] proposes a permissioned blockchain to track plasma. The system records digital tokens representing donors of blood: each blood donation is recorded as a token. This token records many medical testings transforming the initial token. This approach allows doctors to identify the origin of the plasma and minimize the risks of using tainted plasma.

[2] www.gcoin.com/

CHAPTER 10

CONCLUSION AND FUTURE WORK

10.1 Conclusion

This thesis presented *.chain, a framework providing a comprehensive set of software tools for automatising the design and the development of blockchain based SCMSs. This framework allows goods producers, who are experts of their specific domains and production processes, to exploit a user friendly editor and an ad-hoc DSGL for easily representing the main types of supply chains, and –once designed– to translate the supply chain scheme they designed into the solidity smart contracts implementing the blockchain-based SCM system. The framework also automatically produces two web pages for the management of the SCM system: one for the supply chain administrator, and the other for the supply chain participants. Through the administration graphical interfaces, the supply chain administrator registers the participant to the supply chain and assigns them proper roles to grant them a specific set of rights of executing operations. Using the participant graphical interfaces, the supply chain participants registered to the supply chain can easily register the operations they performed on the assets during the production process according to the roles grant them by the supply chain administrator. Finally, the consumer of the final goods, through a third graphical user interface, can easily inspect all the steps of the related production process, that are registered o the blockchain and hence always available and not alterable.

We validated our framework by exploiting it to represent three supply chain use case: the soybean traceability study case, the Protected Designation of Origin olive oil, and the Carne PRI (Pezzata Rossa Italiana) meat cer-

tification. In particular, for each use case, we built the corresponding model introducing role-based authorization controls and also other constraints on operations. Then, we used our framework to translate the supply chain schema into solidity code, and to produce the supply chain Administrator and Participants interfaces. We found out that using our tool to model the three supply chains was quite easy, and does not really require the knowledge of blockchain principles. Moreover, we analyzed through these examples the costs of implementing an SCM System thanks to the framework. We built the model, ran the various tools, and loaded the smart contracts into a local Ethereum test net. Through this, we recorded the costs of individual transactions, calculating different -worse and best- scenarios. For both Olive Oil and Carne PRI, we evaluate the cost of each operations of the supply chain, then the total cost considering the scenario of all the transactions were to be performed. Hence, for each individual asset we would have the maximum cost required by the blockchain. As a matter of results, we identified a relatively low cost to track a single asset using the *-chain framework to create their respectively SCM System.

For this reason, we belief that the *-chain tool could be successfully adopted in a large number of other use cases, thus contributing to the spreading of blockchain adoption in common business problems.

10.2 Future Work

As future work we plan to better analyze the potential of the DSGL, comparing it with several other models and tools, aiming to underline differences or similarities. We want to determine the different costs in terms of: implementation of the SCM System, speed of execution, additional coding needed, and scalability and robustness to the supply chain changes. In addition, is of relevant interest to determine which the best coding techniques are, in order to reduce transaction costs based on the blockchain technologies used. Last, to refine the programming of automatic translation of smart contracts, comparing various models and solutions. Also, we plan to develop a validation phase for the generated smart contract, to ensure a more clear and working output code. Furthermore, it would be interesting to analyze the soundness of the product code solidity, through quality analysis. Our task is to refine the framework and make the graphical interface easier to handle, especially for inexperienced managers who lack specific knowledge of the model. As a successive step we plan to translate it into other languages for DLT, such as Chaincode[1]. Given our the model, coupled with the graphical design interface we must consider that the framework is easily adaptable -upgradable- to

[1] https://hyperledger-fabric.readthedocs.io/en/release-1.3/chaincode.html

different types of blockchain. Once the supply chain schema has been drawn, it generates the scJR. This representation can be translated into various other smart contract programming languages, or DLT strategies: since SCs useful for product traceability need to meet certain constraints in their path, translation into smart contract languages represents the "natural" application. For these reasons, the framework is not tied to the type of blockchain in which the translation is performed. Several plugins to the framework for translation into different languages will be designed. All planned upgrades are inter-independent from each other.

To further improve the usability of the framework, we plan to introduce the possibility of defining macro-functions, i.e., composition of existing operations. The goal is to reduce procedural costs and earn an easier and clearer design.

We are also considering integrating *Self-sovereign identity* (SSI) technology for General Data Protection Regulation (GDPR) compliant[2] identity management, instead of outsourcing the storage of sensitive data to off-chain systems. in the case of GDPR, devolving user management system on the blockchain has an expansice cost; in order to ensure a web-based system following the GDPR guidelines more flexible technologies must be used. Hence, for the management of user profiling, a monolithic data base is required, in order to follow DPSR guidelines.

Finally, we plan to investigate whether the adoption of different data structures to represent the asset history reduces the deployment and execution costs of the proposed solution. Our framework uses a data structure mainly based of a dictionary of two dimensions: one stores the list of generated assets , the other one stores the history of each asset: a more complex data structures may perhaps be less onerous for the smart contracts generated, so for the transactional cost on the blockchain.

[2] `https://gdpr-info.eu/`

BIBLIOGRAPHY

[1] Rita Azzi, Rima Kilany Chamoun, and Maria Sokhn. The power of a blockchain-based supply chain. *Computers & Industrial Engineering*, 135:582–592, 2019.

[2] Gavina Baralla, Simona Ibba, Michele Marchesi, Roberto Tonelli, and Sebastiano Missineo. A blockchain based system to ensure transparency and reliability in food supply chain: Euro-par 2018 international workshops, turin, italy, august 27-28, 2018, revised selected papers, 01 2019.

[3] Stefano Bistarelli, Francesco Faloci, and Paolo Mori. *.chain: automatic coding of smart contracts and user interfaces for supply chains. In *Third International Conference on Blockchain Computing and Applications, BCCA 2021, Tartu, Estonia, November 15-17, 2021*, pages 164–171. IEEE, 2021.

[4] Stefano Bistarelli, Francesco Faloci, and Paolo Mori. Towards a graphical DSL for tracing supply chains on blockchain. In Ricardo Chaves, Dora B. Heras, Aleksandar Ilic, Didem Unat, Rosa M. Badia, Andrea Bracciali, Patrick Diehl, Anshu Dubey, Oh Sangyoon, Stephen L. Scott, and Laura Ricci, editors, *Euro-Par 2021: Parallel Processing Workshops - Euro-Par 2021 International Workshops, Lisbon, Portugal, August 30-31, 2021, Revised Selected Papers*, volume 13098 of *Lecture Notes in Computer Science*, pages 219–229. Springer, 2021.

[5] Stefano Bistarelli, Francesco Faloci, and Paolo Mori. *-chain: A framework for automating the modeling of blockchain based supply chain tracing systems. *Future Gener. Comput. Syst.*, 149:679–700, 2023.

[6] Stefano Bistarelli, Francesco Faloci, Paolo Mori, and Carlo Taticchi. Olive oil as case study for the *-chain platform. In Maurizio Pizzonia

and Andrea Vitaletti, editors, *Proceedings of the 4th Workshop on Distributed Ledger Technology co-located with the Italian Conference on Cybersecurity 2022 (ITASEC 2022), Rome, Italy, June 20, 2022*, volume 3166 of *CEUR Workshop Proceedings*, pages 94–102. CEUR-WS.org, 2022.

[7] Stefano Bistarelli, Francesco Faloci, Paolo Mori, Carlo Taticchi, and Marino Miculan. Modeling carne PRI supply chain with the *-chain platform. In Paolo Mori, Ivan Visconti, and Stefano Bistarelli, editors, *Proceedings of the Fifth Distributed Ledger Technology Workshop (DLT 2023), Bologna, Italy, May 25-26, 2023*, volume 3460 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2023.

[8] Debajyoti Biswas, Hamed Jalali, Amir H. Ansaripoor, and Pietro De Giovanni. Traceability vs. sustainability in supply chains: The implications of blockchain. *Eur. J. Oper. Res.*, 305(1):128–147, 2023.

[9] Kamanashis Biswas, Vallipuram Muthukkumarasamy, and Wee Lum. Blockchain based wine supply chain traceability system, 11 2017.

[10] Louis Brennan and Ruslan Rakhmatullin. Global value chains and smart specialisation strategy. thematic work on the understanding of global value chains and their analysis within the context of smart specialisation., 12 2015.

[11] Miguel Pincheira Caro, Muhammad Salek Ali, Massimo Vecchio, and Raffaele Giaffreda. Blockchain-based traceability in agri-food supply chain management: A practical implementation. *2018 IoT Vertical and Topical Summit on Agriculture - Tuscany (IOT Tuscany)*, pages 1–4, 2018.

[12] Jingjing Chen, Tiefeng Cai, Wenxiu He, Lei Chen, Gang Zhao, Weiwen Zou, and Lingling Guo. A blockchain-driven supply chain finance application for auto retail industry. *Entropy*, 22:95, 01 2020.

[13] Olivia Choudhury, Nolan Rudolph, Issa Sylla, Noor Fairoza, and Amar 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)*, pages 963–970, 2018.

[14] Lanfranco Conte, Alessandra Bendini, Enrico Valli, Paolo Lucci, Sabrina Moret, Alain Maquet, Florence Lacoste, Paul Brereton, Diego Luis García-González, Wenceslao Moreda, and Tullia Gallina Toschi. Olive oil

quality and authenticity: A review of current eu legislation, standards, relevant methods of analyses, their drawbacks and recommendations for the future. *Trends in Food Science & Technology*, 105:483–493, 2020.

[15] Flavio Corradini, Alessandro Marcelletti, Andrea Morichetta, Andrea Polini, Barbara Re, and Francesco Tiezzi. Chorchain: A model-driven framework for choreography-based systems using blockchain. In Andrea Marrella and Daniele Theseider Dupré, editors, *Proceedings of the 1st Italian Forum on Business Process Management co-located with the 19th International Conference of Business Process Management (BPM 2021), Rome, Italy, September 10th, 2021*, volume 2952 of *CEUR Workshop Proceedings*, pages 26–32. CEUR-WS.org, 2021.

[16] Ministero delle Politiche Agricole e Forestali. Olio extravergine fi oliva "terre di siena" di origine protetta. -, - -.

[17] Jan L.G. Dietz. Demo: Towards a discipline of organisation engineering. *European Journal of Operational Research*, 128(2):351–363, 2001. Complex Societal Problems.

[18] Pankaj Dutta, Tsan-Ming Choi, Surabhi Somani, and Richa Butala. Blockchain technology in supply chain operations: Applications, challenges and research opportunities. *Transportation Research Part E: Logistics and Transportation Review*, 142:102067, 2020.

[19] David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Artech House, Inc., USA, 2nd edition, 2007.

[20] Mark Fox, J. Chionglo, and M. Barbuceanu. The integrated supply chain management project, 01 1993.

[21] Ye Gao, Hongwei Gao, Han Xiao, and Fanjun Yao. Vaccine supply chain coordination using blockchain and artificial intelligence technologies. *Comput. Ind. Eng.*, 175:108885, 2023.

[22] Bengang Gong, Huaimiao Zhang, Yiling Gao, and Zhi Liu. Blockchain adoption and channel selection strategies in a competitive remanufacturing supply chain. *Comput. Ind. Eng.*, 175:108829, 2023.

[23] Kannan Govindan, Hamed Soleimani, and Devika Kannan. Reverse logistics and closed loop supply chain: A comprehensive review to explore the future. *Eur. J. Oper. Res.*, 240(3):603–626, 2015.

[24] J. Hao, Y. Sun, and H. Luo. A safe and efficient storage scheme based on blockchain and ipfs for agricultural products tracking. *Journal of Computers (Taiwan)*, 29:158–167, 01 2018.

[25] Martin Holland, Josip Stjepandic, and Christopher Nigischer. Intellectual property protection of 3d print supply chain with blockchain technology. In *2018 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC), Stuttgart, Germany, June 17-20, 2018*, pages 1–8. IEEE, 2018.

[26] Essajide Lhassan, Rachidi Ali, and Fikri Majda. Combining scor and bpmn to support supply chain decision-making of the pharmaceutical wholesaler-distributors. In *2018 4th International Conference on Logistics Operations Management (GOL)*, pages 1–10, 2018.

[27] Zhijie Li, Haoyan Wu, Brian King, Zina Ben Miled, John Wassick, and Jeffrey Tazelaar. A hybrid blockchain ledger for supply chain visibility. In *2018 17th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 118–125, 2018.

[28] Antonios Litke, Dimosthenis Anagnostopoulos, and Theodora Varvarigou. Blockchains for supply chain management: Architectural elements and challenges towards a global scale deployment. *Logistics*, 3:5, 01 2019.

[29] Sidra Malik, Salil S. Kanhere, and Raja Jurdak. Productchain: Scalable blockchain framework to support provenance in supply chains. In *17th IEEE International Symposium on Network Computing and Applications, NCA 2018, Cambridge, MA, USA, November 1-3, 2018*, pages 1–10. IEEE, 2018.

[30] Dianhui Mao, Fan Wang, Yalei Wang, and Zhihao Hao. Visual and user-defined smart contract designing system based on automatic coding. *IEEE Access*, 7:73131–73143, 2019.

[31] Edlira Martiri and Gentjana Muca. DMS-XT: A blockchain-based document management system for secure and intelligent archival. In Endrit Xhina and Klesti Hoxha, editors, *Proceedings of the 3rd International Conference on Recent Trends and Applications in Computer Science and Information Technology, RTA-CSIT 2018, Tirana, Albania, November 23rd - 24th, 2018*, volume 2280 of *CEUR Workshop Proceedings*, pages 70–74. CEUR-WS.org, 2018.

[32] Fredrik Milani, Luciano García-Bañuelos, Svitlana Filipova, and Mariia Markovska. Modelling blockchain-based business processes: a comparative analysis of BPMN vs CMMN. *Bus. Process. Manag. J.*, 27(2):638–657, 2021.

[33] Abubakar Mohammed, Vidyasagar M. Potdar, Mohammed Quaddus, and Wendy Hui. Blockchain adoption in food supply chains: A systematic literature review on enablers, benefits, and barriers. *IEEE Access*, 11:14236–14255, 2023.

[34] Mitsuaki Nakasumi. Information sharing for supply chain management based on block chain technology. In *2017 IEEE 19th Conference on Business Informatics (CBI)*, volume 01, pages 140–149, 2017.

[35] Teijo Peltoniemi and Jarkko Ihalainen. Evaluating blockchain for the governance of the plasma derivatives supply chain: How distributed ledger technology can mitigate plasma supply chain risks. *Blockchain in Healthcare Today*, 2, 05 2019.

[36] Ph.D., Rutner, and Shepherd, C. Is Marketing Still Part of Supply Chain Management, and Should Marketing Academics and Practitioners Care?, 09 2015.

[37] Mehrdokht Pournader. Blockchain applications in supply chains, transport and logistics: a systematic review of the literature. *International Journal of Production Research*, 58, 08 2019.

[38] Daniela Remenska, Jeff Templon, Tim A. C. Willemse, Philip Homburg, Kees Verstoep, Adrian Casajus Ramo, and Henri E. Bal. From UML to process algebra and back: An automated approach to model-checking software design artifacts of concurrent systems. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, volume 7871 of *Lecture Notes in Computer Science*, pages 244–260. Springer, 2013.

[39] Khaled Salah, Nishara Nizamuddin, Raja Jayaraman, and Mohammad Omar. Blockchain-based soybean traceability in agricultural supply chain. *IEEE Access*, 7:73295–73305, 2019.

[40] Affaf Shahid, Ahmad Almogren, Nadeem Javaid, Fahad Ahmad Al-Zahrani, Mansour Zuair, and Masoom Alam. Blockchain-based agri-food supply chain: A complete solution. *IEEE Access*, 8:69230–69243, 2020.

[41] Marek Skotnica, Jan Klicpera, and Robert Pergl. Towards model-driven smart contract systems - code generation and improving expressivity of smart contract modeling. In Guerreiro Sergio, Sousa Pedro, Aveiro David, Guizzardi Giancarlo, Pergl Robert, and Proper Henderik A., editors, *CIAO! Doctoral Consortium, EEWC Forum 2020*, number 2825 in CEUR Workshop Proceedings, pages 1–16, Bolzano, Italy, 03 2021.

[42] Julian Solarte-Rivera, Andrés Vidal-Zemanate, Carlos Cobos, José Alejandro Chamorro-Lopez, and Tomas Velasco. Document management system based on a private blockchain for the support of the judicial embargoes process in colombia. In Raimundas Matulevicius and Remco M. Dijkman, editors, *Advanced Information Systems Engineering Workshops - CAiSE 2018 International Workshops, Tallinn, Estonia, June 11-15, 2018, Proceedings*, volume 316 of *Lecture Notes in Business Information Processing*, pages 126–137. Springer, 2018.

[43] Luca Spalazzi, Francesco Spegni, Alessandra Corneli, and Berardo Naticchia. Blockchain based choreographies: The construction industry case study. *Concurrency and Computation: Practice and Experience*, n/a(n/a):e6740, 2021.

[44] Hartmut Stadtler and Christoph Kilger. *Supply Chain Management and Advanced Planning: Concepts, Models, Software, and Case Studies*. Springer Publishing Company, Incorporated, 4th edition, 2010.

[45] Jayashankar Swaminathan, Stephen Smith, and Norman Sadeh. Modeling supply chain dynamics: A multiagent approach. *Decision Sciences*, 29, 04 2000.

[46] Akyene Tetteh. Supply chain distribution networks: single-, dual- and omni-channel, 05 2014.

[47] John Thornton, John Macarthur, and Husam Barham. Electrification of transport refrigeration units for temperature-sensitive freight: U.s. environmental protection agency region 10 technical assistance case study. *Transportation Research Record Journal of the Transportation Research Board*, 05 2018.

[48] Feng Tian. A supply chain traceability system for food safety based on haccp, blockchain amp; internet of things. In *2017 International Conference on Service Systems and Service Management*, pages 1–6, 2017.

[49] Norbert Trautmann and Cord-Ulrich Fündeling. Supply chain management and advanced planning in the process industries. In Karl-Heinz Waldmann and Ulrike M. Stocker, editors, *Operations Research, Proceedings 2006, Selected Papers of the Annual International Conference of the German Operations ResearchSociety (GOR), Jointly Organized with the Austrian Society of Operations Research (ÖGOR) and the Swiss Society of Operations Research (SVOR), Karlsruhe, Germany, September 6-8, 2006*, pages 503–508, 2006.

[50] Chien-Hua Tsai. Supply chain financing scheme based on blockchain technology from a business application perspective. *Ann. Oper. Res.*, 320(1):441–472, 2023.

[51] Daniel Tse, Bowen Zhang, Yuchen Yang, Chenli Cheng, and Haoran Mu. Blockchain application in food supply information security. In *2017 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, pages 1357–1361, 2017.

[52] Jen-Hung Tseng, Yen-Chih Liao, Bin Chong, and Shih-wei Liao. Governance on the drug supply chain via gcoin blockchain. *International Journal of Environmental Research and Public Health*, 15:1055, 05 2018.

[53] Matthijs [van Bergen], Michiel Steeman, Matthew Reindorp, and Luca Gelsomino. Supply chain finance schemes in the procurement of agricultural products. *Journal of Purchasing and Supply Management*, 25(2):172 – 184, 2019. Supply Chain Finance: Historical Foundations, Current Research, Future Developments.

[54] Durk-Jouke van der Zee and Jack G. A. J. van der Vorst. A modeling framework for supply chain simulation: Opportunities for improved decision making. *Decis. Sci.*, 36(1):65–95, 2005.

[55] Mark von Rosing, Stephen White, Fred Cummins, and Henk de Man. Business process model and notation - BPMN. In Mark von Rosing, Henrik von Scheel, and August-Wilhelm Scheer, editors, *The Complete Business Process Handbook: Body of Knowledge from Process Modeling to BPM, Volume I*, pages 429–453. Morgan Kaufmann/Elsevier, 2015.

[56] Shangping Wang, Yinglong Zhang, and Yaling Zhang. A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems. *IEEE Access*, 6:38437–38450, 2018.

[57] Ting Wang, Qiang Lan, and Yong Chu. Supply chain financing model: Based on china's agricultural products supply chain. *Applied Mechanics and Materials*, 380-384:4417 – 4421, 2013.

[58] M. Westerkamp, F. Victor, and A. Küpper. Blockchain-based supply chain traceability: Token recipes model manufacturing processes, 2018.

[59] Maximilian Wöhrer and Uwe Zdun. Domain specific language for smart contract development. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9, 2020.

[60] Zhe Yang, Kan Zheng, Kan Yang, and Victor C. M. Leung. A blockchain-based reputation system for data credibility assessment in

vehicular networks. In *28th IEEE Annual International Symposium on Personal, Indoor, and Mobile Radio Communications, PIMRC 2017, Montreal, QC, Canada, October 8-13, 2017*, pages 1–5. IEEE, 2017.