Future Generation Computer Systems 149 (2023) 679–700

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

*-chain: A framework for automating the modeling of blockchain based supply chain tracing systems

Stefano Bistarelli [a], Francesco Faloci [b,c,*], Paolo Mori [c]

[a] Universita' degli studi di Perugia, Perugia, Italy
[b] Universita' di Camerino, Camerino, Italy
[c] Istituto di Informatica e Telematica - Consiglio Nazionale delle Ricerche, Pisa, Italy

ARTICLE INFO

Article history:
Received 4 May 2022
Received in revised form 16 May 2023
Accepted 10 July 2023
Available online 20 August 2023

Keywords:
Supply chain
Blockchain
Distributed ledger technology
Domain specific graphical language
Smart contracts
Automatic smart contract generation

ABSTRACT

Nowadays, creating a blockchain-based system for supply chain tracing is a complex task. This paper defines a model, a graphical domain specific language, and a set of tools aimed at helping supply chain domain experts to create blockchain based tracing systems for their supply chains. Starting from a graphical representation of the supply chain, the solidity smart contracts implementing the related tracing system are automatically generated by our framework. Small interventions of programmers are required to customize and finalize such smart contracts. A set of web based interfaces to interact with such smart contracts are also automatically generated. We are confident that our results will increase blockchain usage for supply chain traceability thanks to the automatic process of smart contract generation.

1. Introduction

A Supply Chain (SC[1]) has been defined in [1] as an organizations network 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 into intermediate products/services and then into the final one.

In order to give a more precise definition of an SC, it is necessary to understand which aspects of the products/services and which operations are under analysis, the type of process represented, and the actors operating in the network. Regardless of the specific process represented by the SC, having the capability of automatically tracing and controlling the actions performed by the actors of the SC is of paramount importance to obtain an effective and efficient process. *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 [2,3]. A number of SCM system 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.

A number of blockchain-based solutions tailored for SCs [4–7] are already available but, according to the researchers proposing them, such solutions are still focused on a specific scenario and very difficult to be generalized. Consequently, to promote the usage of blockchain technology for SC tracing and with the aim to make the SCM systems development easier and usable by supply chain domain experts, who are not necessarily distributed ledger technology experts as well, we propose the *-*chain* framework.

**-chain* is a framework that helps supply chain domain experts to design an SCM system and to automatically obtain the smart contracts implementing it 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 *Domain Specific Language (DSL)* representing such model; and *(iii)* a *set of tools* able to implement frontend and backend of a blockchain based SCM system, starting just from the SC graphical representation.

As a matter of fact, *-chain provides a graphical interface that is used by supply chain domain experts to represent their supply chain processes in terms of assets and operations performed

* Corresponding author at: Universita' di Camerino, Camerino, Italy.
E-mail address: francesco.faloci@unicam.it (F. Faloci).

[1] A list of all the acronyms used in the paper is available in Appendix.

[2] www.swarmnetwork.org/.

[3] www.ripe.io.

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

https://doi.org/10.1016/j.future.2023.07.012

on them. Such operations can be defined according to a set of predefined types, and they can be customized by configuring constraints on the properties of the involved assets, and authorization rules on the role of users executing them. Starting from the graphical representation, 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. Such smart contracts are then finalized (when necessary) by developers according to the specific needs of the supply chain. Starting from the graphical representation of the supply chain as well, the suite of tools also creates the web interfaces to interact with the SCM system, namely the Supply Chain Administration Interface and the Supply Chain Participant Interface. The former will be used by the supply chain administrator to manage supply chain participants (e.g., registration and roles), while the latter will be used by the supply chain participants to invoke smart contracts and record the operations executed on the SC assets.

The original contributions brought by this paper are the following:

- A general model to represent most of supply chains (sales, distribution, production), and the related DSL;
- A tool providing a graphical interface to design specific supply chains through the previously defined DSL;
- A software suite that, starting from a specific supply chain represented through our DSL, produces:
  - the skeleton of the smart contracts implementing the related SCM system;
  - a graphical interface enabling supply chain administrators to define the users allowed to interact with the smart contracts, their roles and their rights;
  - a graphical interface connected with the created smart contracts enabling authorized supply chain participants to permanently record on the blockchain the actions they executed on the assets of the SC.

This paper extends and supersedes our previous works [8,9]. In particular, in [8] we presented the initial idea of the supply chain representation model and of the DSL, which are then exploited in [9] to represent a small excerpt of a well known supply chain. This paper presents several enhancements with respect to our previous works. First of all, we refined the proposed model and DSL by adding some new operations in order to allow a better representation and differentiation of the change of owner and of controller of an asset. We also refined the representation of the authorization rules that are now paired with operations to regulate their execution by supply chain participants. Another relevant contribution provided by this paper w.r.t. our previous works is the definition of the complete workflow of the creation of SCM systems. In order to validate the proposed framework, in this paper we give a complete representation of the well known supply chain we introduced in [9] as reference example. Finally, in this paper we also present an experimental evaluation of the deployment and execution costs of SCM systems produced with our framework.

The rest of the paper is organized as follows: Section 2 introduces the blockchain technology, the solidity language, the SCM systems, and the reference example we have already used in [8]. Section 3 presents our approach and the proposed framework architecture. Section 4 describes the model we defined and the related DSL. Section 5 shows the representation of the supply chain of the reference example using our DSL. Section 6 describes the graphical interface allowing supply chain domain experts to create their SCM systems, and presents the graphical interfaces produced by our framework allowing supply chain administrators and participants to interact with the produced SCM systems. Section 7 provides some details about the process that, starting from the SC graphical representation, produces the smart contracts implementing the SCM system. Section 8 presents an experimental evaluation of the deployment and execution costs of such smart contracts. Section 9 presents some related works, while Section 10 draws the conclusions and describes possible future works.

2. Background

2.1. 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. A Blockchain is a sequence of blocks connected through cryptographically 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 and participate to the consensus algorithm to create a new block including them. Blockchain main features are decentralization, immutability, and transparency. First generation blockchains were focused on cryptocurrencies (such as Bitcoin[5]). Subsequently, a new generation of blockchains, able to execute programs called smart contracts, were designed. The most famous examples are Ethereum[6] and EOS.IO.[7] The advent of smart contracts paved the way for blockchain based applications in industry and in public sectors [4,5,10]. Smart contracts are "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[8] which is widely used on the Ethereum blockchain, and we use it in the following of this paper as our reference language.

2.2. 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 SCM system 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 SCM system 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 SCM system has actually been executed. For instance, in case of blockchain-based SCM systems, the Blockchain Oracle technology [11] (and, in particular, the Chainlink[9] implementation) could be exploited to collect in a reliable way and store on the SCM system, information about some relevant physical features of the assets during the production process which prove the execution of some operations. Notice

[5] bitcoin.org.
[6] ethereum.org/.
[7] eos.io/.
[8] docs.soliditylang.org/en/v0.8.4/.
[9] https://chain.link/.

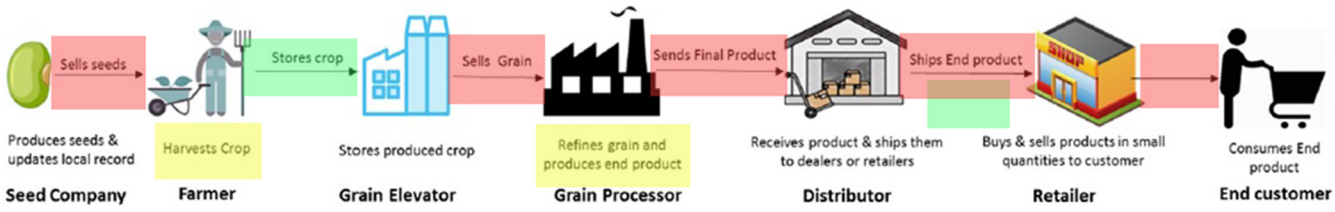


Fig. 1. Supply chain of the soybeans reference example [7].

also that, when two parties are involved in the same operation, the SCM system could require that they both confirm that such operation has been executed. For instance, in our framework we have chosen to have pair of operations when one supply chain participant transfers the ownership or the control of an asset to another (that is, the pairs of methods sell/buy and giveControl/takeControl, see Section 4). In this way, the operation recorded on blockchain-based SCM system is confirmed by both the involved parties.

Several types of SCs have been defined in the scientific literature [12–15], 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.

2.3. Reference example

Fig. 1 shows an example of supply chain use case representing the soybeans life cycle, from the seed production to the end customer, as described in [7]. The leftmost green bean in the figure represents the first asset of the supply chain (*Seed*). In this example, subjects must have the role *Seed Company* to be allowed to create *Seed* assets. The first operation done on the *Seed* asset is *Sell*. This operation represents the change of the owner from the entity who created the asset to another entity with the role *Farmer*. The second operation of the supply chain is *Harvest Crop*, it is performed by the farmer who bought the seeds, and it represents a transformation of the *Seed* assets into other assets: the *Crops*. The harvested *Crops* are then gathered and stored in an elevator, waiting to be sold. The farmer sells the Grain stored in the Grain Elevator to a Grain Processor (a refinery or industry suitable for the purpose) where the Grain undergoes a transformation process (called *Refines grain and produces end product* in the figure) until it becomes the final product ready to be sold. The final product is then sold by the refiner to the main distributor. In this phase, in addition to the sale, we can see how the product is stored in warehouses and then distributed in the reference distribution network. The final destination of the refined product is the single store, having the role *Retailer*, where the final product is available for sale to the customer.

The described example involve several class of operation: a move transaction (green background) changing the physical placement of an asset, the sale (pink background) changing the owner of an asset, and some transformation phases (yellow background) changing some properties of the asset or destroying it and creating new ones.

3. The *-chain approach

The *-chain framework is aimed at driving its users in the design and development of SCM systems for tracing their production processes exploiting the blockchain technology.

The main idea underlying the *-chain approach is to enable the experts of a specific supply chain process (e.g., the soybeans production process described in Section 2.3) 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 executed by an expert of the specific supply chain process, while the related smart contracts programming will be executed by developers expert in blockchain technology.

The *-chain framework consists of the components in Fig. 2:

- The supply chain General Model (scGM): a general model and the related Domain Specific graphical Language (DSL) allowing to represent the most common types of supply chains. The scGM and the DSL are described in detail in Section 4;
- The supply chain Design Interface (scDI): a graphical editor allowing the supply chain domain experts to design the SCM systems representing their specific supply chains using the aforementioned DSL. 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 scDI is described in detail in Section 6.1;
- 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 functions for tracing the execution of all the operations supported by the asset it represents. The scMT is described in detail in Section 7;
- 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 two graphical 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 6.2;
  - 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 6.3.

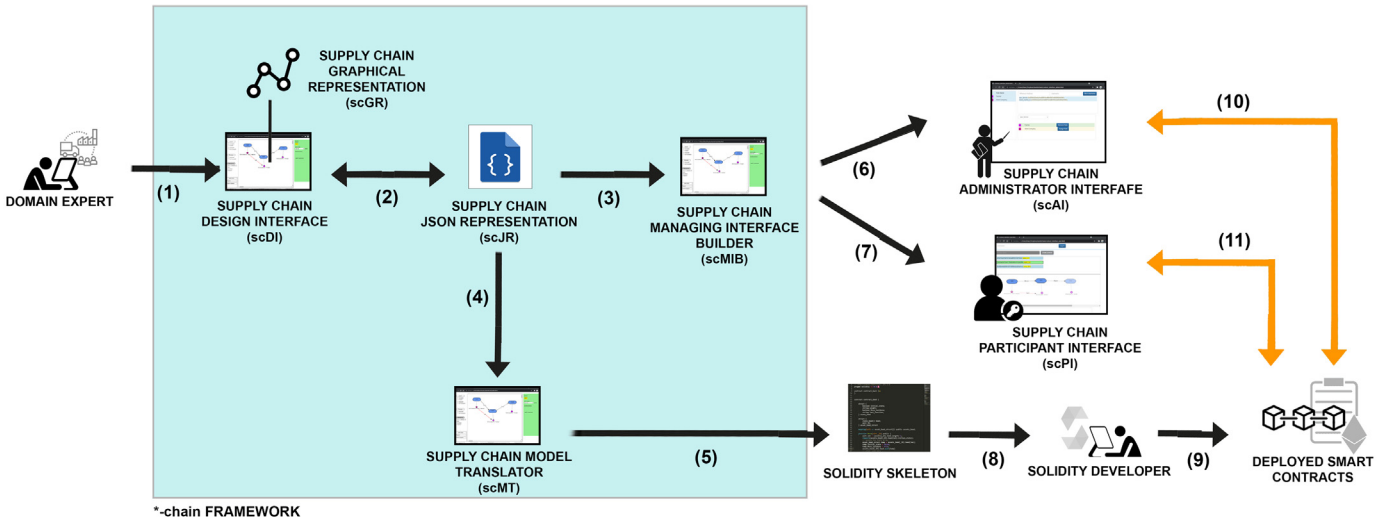


Fig. 2. *-chain Framework usage workflow.

Fig. 2 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 supply chain domain expert uses the scDI (step 1) to design its supply chain process using the DSL. The result of the design step is the scJR (step 2) which is passed by the scMT (step 4) to produce the skeleton of the smart contracts that will implement the SCM system (step 5). This skeleton will be passed to the Solidity Developers (step 8), who will add the code for managing specific requirements thus obtaining the final smart contracts that will be deployed on the blockchain (step 9). Moreover, the domain expert uses the scMIB (step 3) to automatically produce the graphical interfaces to be made available to supply chain Administrator and to the supply chain participants (respectively, steps 6 and 7). These two interfaces communicate directly with the SCM system smart contracts deployed on the blockchain (steps 10 and 11) through specific functions.

Please notice that, although the supply chain Graphical Representation describing all the possible flows of the production process is defined by the supply chain domain expert, the execution of the production process operations and the calls to the associated smart contract methods of the SCM system are performed by the actors of such process. Hence, the execution is decentralized because the operations representing the steps of the production process are carried out by distinct actors, the supply chain participants, which use the SCM system to keep track of which operations they performed on which assets. In addition, notice that the proposed tracing system also allows to dynamically define the actors executing the production process steps. As a matter of fact, the same operation of the production process could be carried out by distinct actors for distinct instances of each asset. For instance, the farmer could send distinct lots of crops to distinct grain processors, and this can be naturally represented in our model and traced in our SCM system.

Notice also that, deploying the SCM system on a public blockchain could somehow be seen as a disclosure of the production processes to the competitors. However, the operations that are tracked in the SCM system are typically quite high level, and each actor of the supply chain does not reveal on the blockchain the details of the work they carried out to execute each operation.

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

4. A general model and a domain specific graphical language for representing supply chains

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. Our framework is addressed to supply chain domain experts, who are supposed not to know how to program and use blockchain technologies. For this reason, it provides a user-friendly graphical editor allowing supply chain domain experts to simply design their supply chains through our graphical language. This would relieve supply chain domain experts from the burden of learning a textual programming language just for representing their supply chains. Moreover, the graphical editor prevents its users from possible syntactical errors, which are typical of using a textual interface.

Notice that, differently from other approaches in the scientific literature (for instance see [16–18], and [19]) which are based on Business Process Modeling Notation, BPMN [20], we have chosen to define a customized model since our approach is asset-based (instead of process-based like BPMN architecture). Indeed, a relevant difference with the BPMN approach is that in our model we first define the assets that are involved in the production process, along with the properties that characterize them, and then we define the operations that can be executed on each asset during the production process, indicating whether these operations simply involve a modification of the properties of the asset, or they transform an asset into another asset. Hence, the supply chain is defined starting from this set of assets, by properly connecting them through a set of operations that represent the workings and the transformations that are executed during the production process. The architecture of the blockchain-based SCM system produced with our approach is asset-based as well. As a matter of fact, each asset is represented with a specific smart contract, which defines all the possible operations that can be executed on such asset during the production process and the effects they have on it. At the same time, this smart contract keeps track of all the operations that have been actually executed on the asset it represents, of the values of the related properties, and of their changes as a consequence of such operations. Summarizing, the proposed approach differs from the BPMN-based ones because the supply chain is defined starting from the involved assets, and defining operations on them, while BPMN-based approaches,

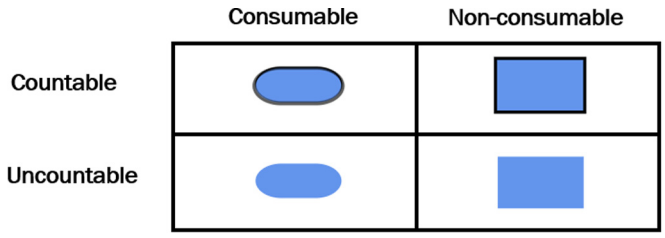


Fig. 3. Graphical representation of assets.

instead, define the supply chain starting from the flow of operations that define the production process. Hence, the difference is somehow similarly to the one between procedural and object-oriented programming languages, where both of them can be used to solve the same problem, but they have two distinct views of the problem.

In our model, we identify three families of elements that will be the bricks of any SC: *Assets, Containers, and Operations*. In addition, also *Roles & Rights* are considered to perform authorizations on SC operations. Starting from the general model, we defined the related DSL by giving a graphical representation of the aforementioned elements. The DSL is then used by the *-chain framework users, i.e., the supply chain domains experts, to represent their specific supply chains. For instance, Section 5 shows how the soybeans supply chain reference example can be represented using our DSL.

In the following of this section, we give a detailed description of the elements composing the general model and of the corresponding graphical representations.

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., the oil asset needs to be stored in oil tins or bottles). On the opposite, an asset is *countable* when it can be tracked without the need of containers and when it can easily be identified (e.g., a car or a cow). 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 consumable assets because when they turn to *Plantation* (after some time), they do not exist anymore. Fig. 3 shows the graphical representation of asset types. For instance, a countable and non-consumable asset is represented by a rectangular box with borders.

Each asset has a number of properties which characterize it. A set of default properties, that are common to all the assets, are automatically paired to each asset when it is created. In the following of this paper we assume that the default properties are: *Owner*, *Controller*, and *Position*. Since these properties are always paired to all assets, for the sake of clarity (in particular in case of complex supply chains), they are not explicitly represented in the scGR. The *Owner* property stores the ID of the supply chain participant who has the ownership of the asset (e.g., the owner bought that asset from another participant or created it). The *controller* property, instead, stores the ID of 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, 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.

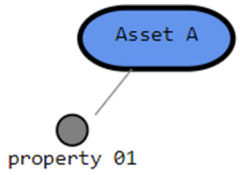


Fig. 4. Example of asset with its asset specific property.

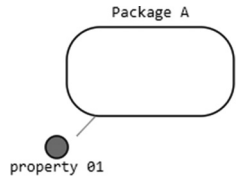


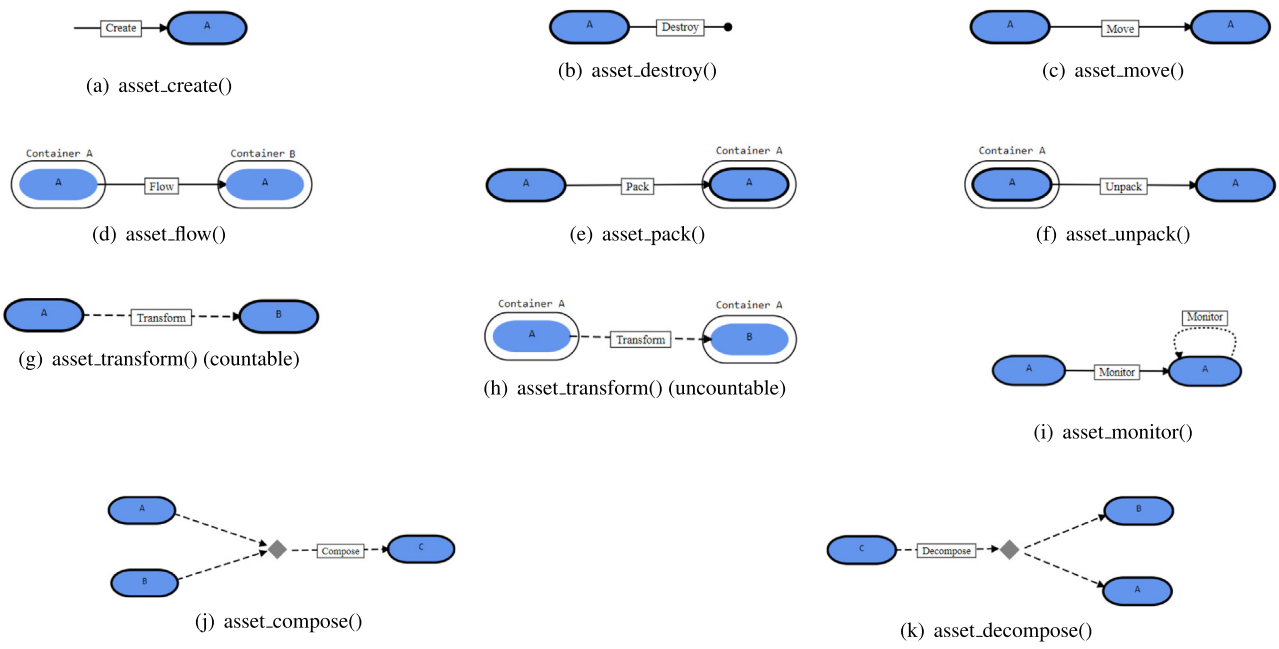
Fig. 5. Example of container with its property.

Besides the default ones, 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 only one asset specific property called *weight*. In our DSL, the asset specific properties are represented in the scGR through small circles connected to their asset by an edge (as shown in Fig. 4). In the following of this section, we will specify, for each operation of our model, when the asset specific properties are shown in the scGR.

Containers are the components of our model representing what can contain – usually uncountable – assets. 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. Similarly to assets, containers can be consumable or non-consumable. Fig. 5 shows a simple example of consumable container (called *Package A*) with a property (called *property 01*). 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. Containers are usually countable and traceable objects. In our DSL, 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 (like the one in Fig. 5). Moreover, to represent that an asset is placed into a container, the box representing the asset is drawn inside the box representing the container.

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 user is the one of registering on the blockchain that such user declares to have executed the corresponding operation on the real asset. An operation executed on an asset (except for the sell one) can be registered on the SCM system only by the user 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 and unique mnemonic name (having the asset name as prefix). 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



(a) asset_create() (b) asset_destroy() (c) asset_move()

(d) asset_flow() (e) asset_pack() (f) asset_unpack()

(g) asset_transform() (countable) (h) asset_transform() (uncountable) (i) asset_monitor()

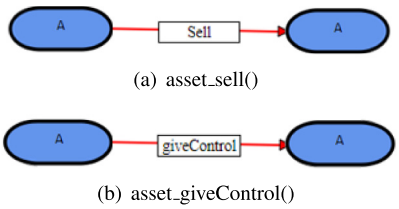
(j) asset_compose() (k) asset_decompose()

Fig. 6. Main operations defined in our model.

for each asset, and since they have the same meaning for all the assets, such operations maintain their original names.

Our DSL represents each operation type with a distinct graphical format, as shown in Fig. 6, 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* (Fig. 6(a)): this operation is used to create a new asset. In the scGR, all the asset specific properties of the new asset must be explicitly represented, and the supply chain participant who invokes the operation must provide the values of all such properties, to be recorded in the SCM system. The default properties of the asset, instead, are not represented in the scGR, because the framework knows how to deal with them. For instance, the owner and the controller properties of an asset are initialized with the id of the supply chain participant who invokes the *asset_create* operation.
- *asset_destroy* (Fig. 6(b)): this operations is applied to destroy an existing asset. When a destroy operation is executed, no further operations can be executed on that asset.
- *asset_move* (Fig. 6(c)): this operation is invoked by a supply chain participant to update of the physical position of the asset. Since this operation is exactly aimed at modifying the *position* default property of the asset, it is not necessary to explicitly represent such property for this operation in the scGR.
- *asset_pack* (Fig. 6(e)): this operation represents the packing of assets into a container. All kinds of assets (countable/uncountable and consumable/non-consumable) can be stored in containers (for uncountable is mandatory, while for countable is discretionary). This operation does not change the original asset information. The *asset_pack* operation is used also to represent the packing of a container (embedding an asset) into another container. Actually, this operation can be repeated several times, to represent that containers have been iteratively packed inside other containers. In the scGR, the asset specific properties of the container are explicitly represented.
- *asset_unpack* (Fig. 6(f)): this operation represents the unpacking of assets from a container. When a container has been packed into another container, this operation is used also to represent the unpacking of the inner container form the outer one. Consumable containers are destroyed by the unpack operation, while non-consumable ones become empty.
- *asset_flow* (Fig. 6(d)): this operation is used when an asset is moved from one container into another. It can be seen as the sequential execution of an *asset_unpack* and an *asset_pack* operations. Asset_flow is often used with uncountable assets. In the scGR, the asset specific properties of the new container are explicitly represented.
- *asset_transform* (Figs. 6(g) and 6(h)): the transformation operations are meant to represent the execution of operations which modify the asset features. 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). Since a new asset is actually created as a consequence of the *asset_transform* operation, in the scGR all the asset specific properties of the new asset are explicitly represented. When the *asset_transform* operation is invoked



(a) asset_sell()

(b) asset_giveControl()

Fig. 7. Sell and giveControl operations.

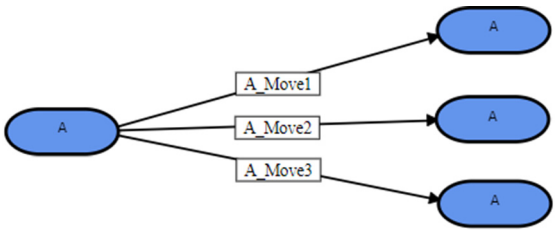


Fig. 8. Alternative operations defined on an asset.

by a supply chain participant, the latter must provide the values of such properties, to be stored on the SCM system. Instead, the default properties of the new asset are not shown in the scGR, because the framework knows them and how to collect their values. For instance, the owner and the controller of the newly created asset are set to be the same as the older one. In our reference example, a quantity of seeds represented with an uncountable consumable asset (*Seed*) inside a container (*Sack*) is modified by a transform operation (*Seed_Planting*) causing the destruction of the *Seed* asset and the creation of the *Plantation* asset representing a portion of the cultivated field inside the container *Field*.

- *asset_monitor* (Fig. 6(i)): the monitor operation is aimed at recording on the SCM system 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. This operation is used also to record on the SCM system a modification of the properties of the asset. In the scGR, the right instance of the monitored asset (the one with the incoming arrows) reports (only) the properties that are updated by the operation.
- *asset_compose* and *asset_decompose* (Figs. 6(j) and 6(k)): the compose operation creates a new asset using existing assets having the same owners without destroying them. The representation in the scGR of asset properties (both asset specific and default ones) and the initialization of their values for the *asset_compose* operation is managed in the same way as for the *asset_transform* operation.
  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.

- *asset_sell* (Fig. 7(a)): 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”. Moreover, since this operation is exactly aimed at changing the value of the *owner* default property of the asset, this property is not explicitly represented in the scGR.
- *asset_giveControl* (Fig. 7(b)): 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. 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”. Moreover, since it is clear that the only aim of this operation is to change the value of the default asset property *controller*, this property is not explicitly represented in the scGR.

Notice that, although all the operations in Figs. 6 and 7 are applied over consumable assets (and containers), the same operations are also applicable to non-consumable ones.

Also notice that more than one outcoming arrow can be originated from the box representing an asset, as shown in Fig. 8, meaning that, at a given point of the supply chain, the supply chain participant having the control of an asset can choose among a number of (distinct) operations to be executed on such asset. Symmetrically, more than one incoming arrow can enter a box representing an asset, meaning that such asset can be obtained following two or more distinct production processes. This would also allow the supply chain domain expert to design supply chains having operations loops. In this case, since it is always the controller of the asset who decides the operation to be performed on the asset among the one admitted in the current state of the asset, is the controller who decides whether to exit or not from an operation loop.

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 5) 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.

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. The idea is that each operation executed on an asset requires a specific capability from the supply chain participant executing it. For instance, the actor who plants seed in a field must be a farmer. Hence, the SCM system must somehow check that the supply chain participant trying to register the execution of an operation was actually

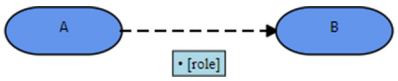


Fig. 9. Authorization rule paired to a transform operation.

entitled to perform such operation, in order to protect the quality of the production process and hence of the final product. For this reason, our model adopts the Role-Based Access Control model (RBAC) [21] 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. Each role represents the capability to perform one or more operations of the supply chain. For instance, the role *Farmer* represents the capability of performing a number of operations such as: planting seeds in a field, harvesting crops from the field, or baling crops. Assigning a role to a supply chain participant, the supply chain domain expert recognizes to such participant the expertise paired to such role. Hence, an authorization rule paired with an operation defines which role is required to be allowed to perform such operation. 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.

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 Fig. 9.

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 6.2). 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.

5. Representing a real case with the *-chain model

In this section we apply our model to represent the soybean supply chain described in Section 2.3. We suppose that the Soybeans Producer (*SBP*) wants to track the soybeans production process, from the seeds acquisition to the soybeans commercialization.

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.

Fig. 10 shows 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 Fig. 10(a). 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 (labeled in the scGR with the name *Seed_create*), which also creates the *Sack* container for the *Seed* asset. As explained in Section 4, 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.

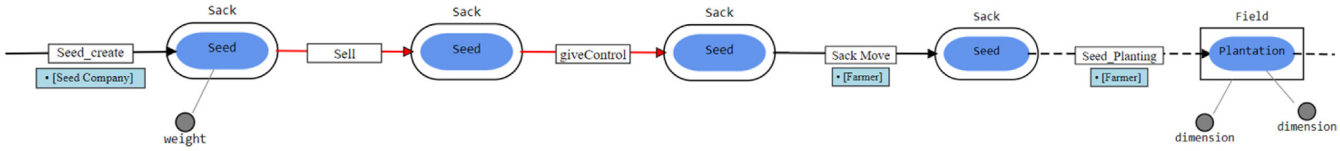
The first operation performed on the *Seed* asset is the *Sell* one. Hence, a second instance of the *Seed* asset (enclosed in the *Sack* container as well) is present in the scGR on the right of the first one, and these two instances are connected through the arrow representing the *Sell* operation.

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.

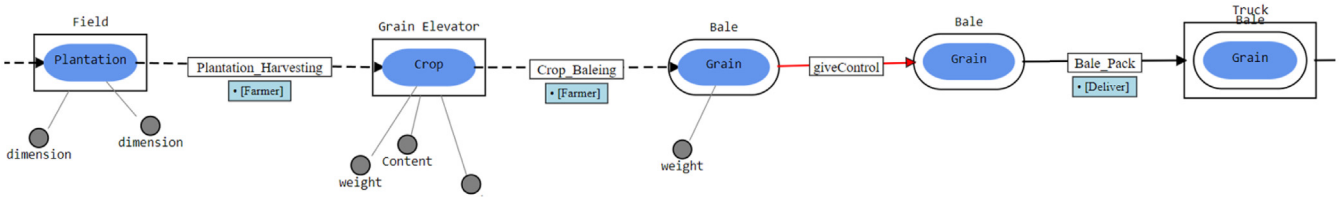
The *giveControl* operation is performed just after the *Sell* one, in order to register on the SCM system that another subject has taken the physical availability of the asset.

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). Hence, 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*.

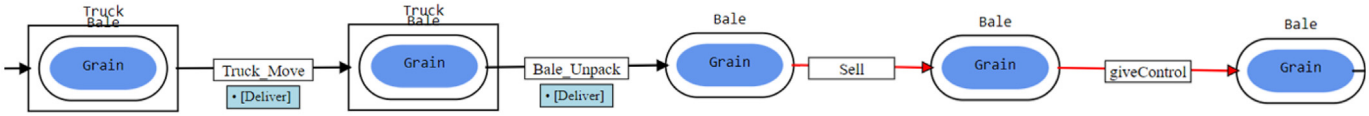
The fourth operation in our reference example supply chain is the *Seed_Planting* one, which is performed on the *Seed* asset of a *Sack*, transforming it in a *Plantation* asset, which represents the specific area of the field where such seeds have been planted. Since the *Seed* asset and the *Sack* 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.



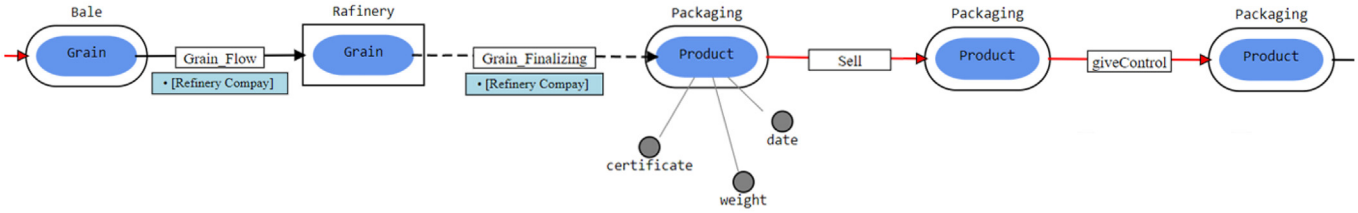
(a) Part I of the supply chain.



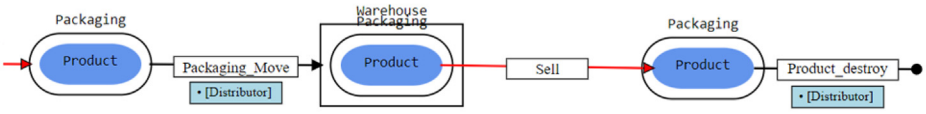
(b) Part II of the supply chain.



(c) Part III of the supply chain.



(d) Part IV of the supply chain.



(e) Part V of the supply chain.

Fig. 10. Representation of the soybeans supply chain [7] with the proposed model.

The fifth operation of our reference example supply chain is *Plantation_Harvesting*, which is executed on the *Plantation* asset (shown in Fig. 10(b)). 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 6.1).

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 Figs. 10(c), 10(d), and 10(e).

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.

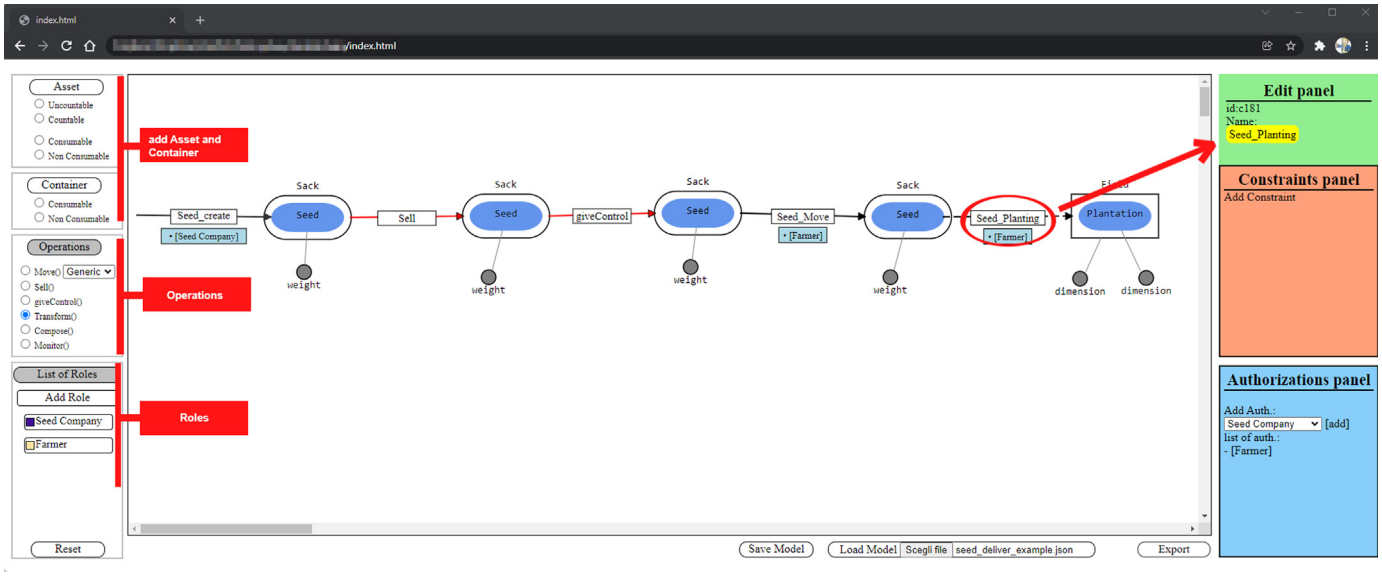


Fig. 11. Example of supply chain Design Interface (scDI).

6. *-chain framework interfaces

As anticipated in Section 3, the *-chain framework provides to its users three user-friendly graphical environments: *(i)* the Supply Chain Design Interface, *(ii)* the Supply Chain Administration Interface, and *(iii)* the Supply Chain Participant Interface.

6.1. Supply Chain Design Interface

The supply chain Design Interface (scDI) is 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 Fig. 2), using the model and the DSL described in Section 4. A screenshot of the scDI is shown in Fig. 11. This web interface 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: Assets, Containers, Operations, and Roles & Rights.

The scDI implementation 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[10] package to build and manage the graphical aspect, while the basic functions of JQuery[11] 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).

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 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.

The operation construct defined in Section 4 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. 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).

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 owner of such asset, and all the other operations can be performed only by the controller of the asset. As a matter of fact, the owner of an asset is the only entity entitled to sell the asset (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 (register on the SCM system) 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.

Through the *Save Model* and the *Export* buttons (placed in the bottom of the scDI), two digital representations of the scGR drawn

10 www.jointjs.com/.
11 jquery.com/.

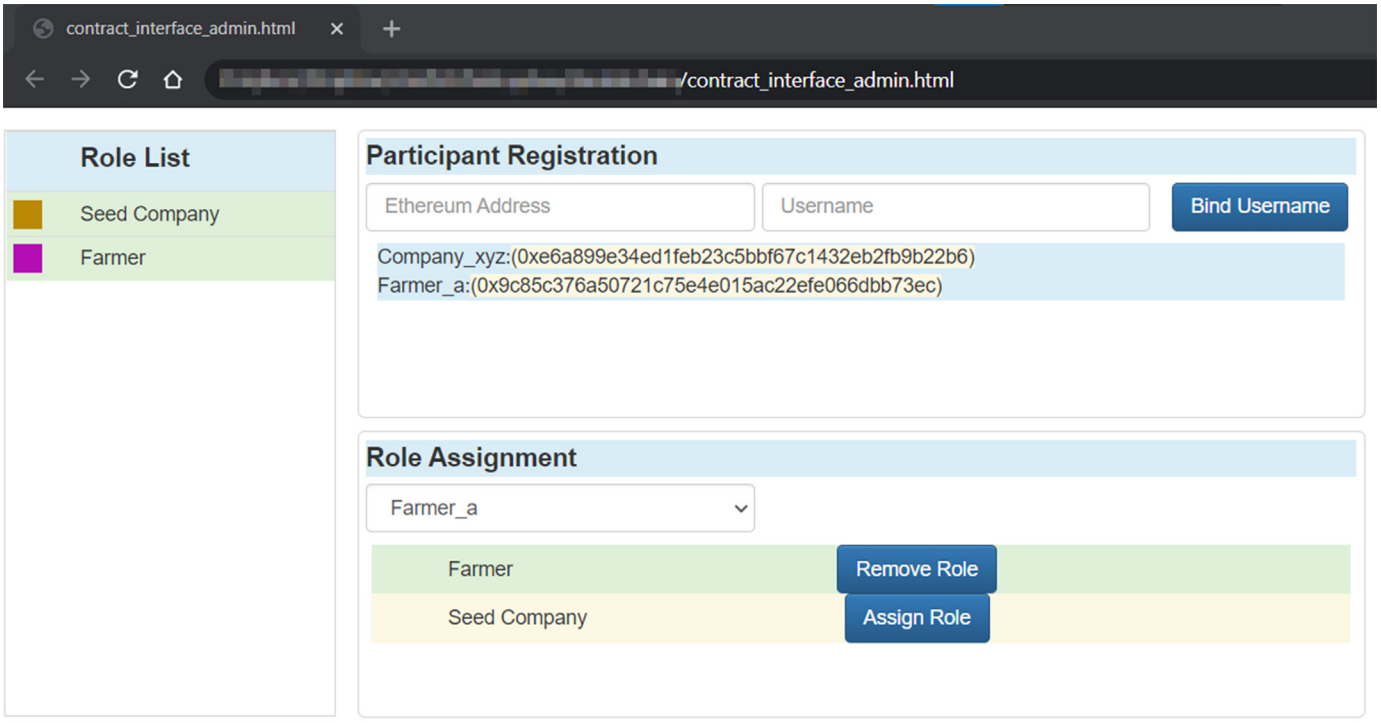


Fig. 12. Example of supply chain Administrator Interface (scAI).

in the central panel are built: the scJR and the Solidity/web3js[12] 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 Fig. 2). The scJR is a *JSON* formatted string consisting of six sections, namely: "*graph*", "*asset*", "*operation*", "*property*", "*constraints*", and "*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. The "*operation*" set lists all the operations occurring in the scGR, i.e., all the arrows connecting couples of assets. 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. The function *Export*, instead, creates two different outputs: the solidity code 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 Fig. 2. Moreover, the scJR is also used by the Supply Chain Managing Interface Builder to create the Supply Chain Administration Interface (step 6 of Fig. 2) and the Supply Chain Participant Interface (step 7 of Fig. 2). In the following, we describe in details such components.

6.2. Supply chain Administration Interface

The supply chain Administration interface (scAI), shown in Fig. 12, is automatically generated by the export function invoked from the scDI, as shown in Fig. 2 (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: Role List, Participant Registration, and Role Assignment. The Role List section 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. These are the users 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 of the available roles to that user. 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 invoked in order to delete a previous role assignment.

6.3. Supply chain participant interface

The supply chain Participant Interface (scPI), shown in Fig. 13, is automatically generated by the export function invoked from the scDI, as shown in Fig. 2 (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. The scPI consists of one main panel which shows the list of assets of which the supply chain participant is the 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

12 web3js.readthedocs.io/en/v1.3.4/.

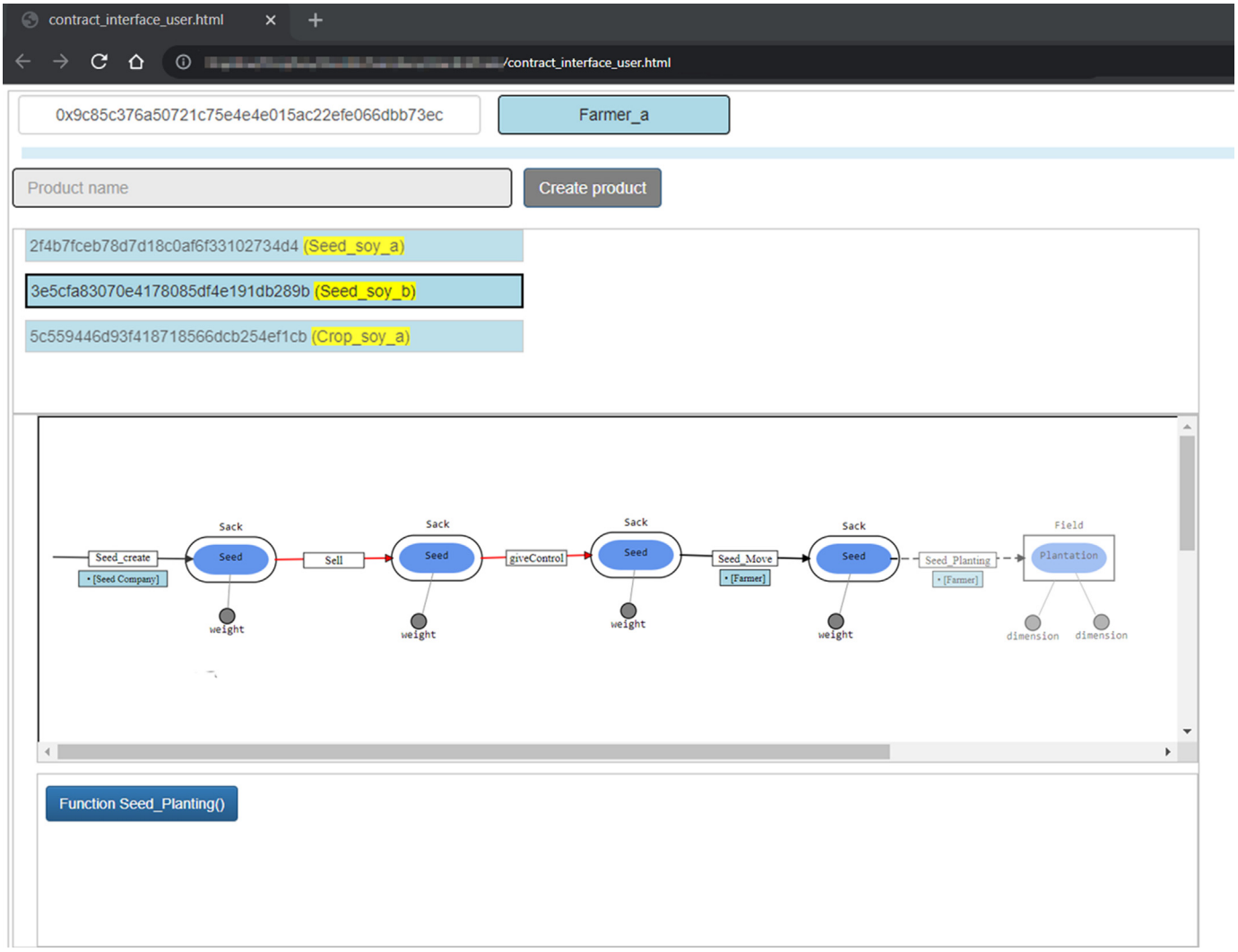


Fig. 13. Example of supply chain Participant Interface (scPI).

the participant has the right to perform on the selected asset, according with the participant's role and the asset status.

Fig. 13 shows an example concerning the user "Farmer_a". The assets shown in the top panel are those for which Farmer_a is the controller or the owner. Since we selected the asset "Seed_soy_b", 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 Farmer_a to claim the execution of the *Seed_Planting* operation on this asset, which is the only operation that can be executed on that asset in its current state.

7. Supply chain model translator

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 Fig. 2).

Fig. 14 shows a simple example of scGR and Fig. 15 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":"c23") which represents the move operation having name *A_Move*; in the *property* section there is one entry (identified by "id":"c46") representing the property called *property 01* of the asset identified by "id":"c8", i.e., the first state of asset *A*.

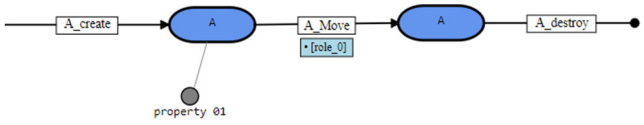


Fig. 14. Example of supply chain Graphical Representation (scGR).

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[13] (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

[13] openzeppelin.com/contracts/.

```
{
  "graph": {…},
  "asset": [
    {
      "id":"c8",
      "name":"A",
      "class":"asset",
      "type":"21"
    },
    {
      "id":"c16",
      "name":"A",
      "class":"asset",
      "type":"21"
    }
  ],
  "operation": [
    {
      "id":"c23",
      "name":"A_Move",
      "class":"operation",
      "type":"move",
      "source":"c8",
      "target":"c16",
      "roles": [
        "role_0"
      ],
      "constraint": [
      ],
      "selor": [
      ]
    },
    {…},
    {…}
  ],
  "property": [
    {
      "id":"c46",
      "name":"property 01",
      "fieldof":"c8"
    }
  ],
  "roles": [
    "role_0"
  ],
  "owners": [1],
  "constraints": [
  ]
}
```


Fig. 15. scJR translation of the scGR in Fig. 14.

that have been executed on the asset. Fig. 17 shows a simple example of smart contract for the asset *A* derived from the scJR shown in Fig. 15. 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 Fig. 14) 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 Fig. 14). 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. 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.

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 Fig. 15, which refers to the scGR in Fig. 14, only one asset is found, asset *A*. The smart contract that is produced by the scMT for asset *A* is shown in Fig. 17. Then, the other two procedures are executed 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 "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 Fig. 14 only one property, *property 01*, is paired with asset *A*, we

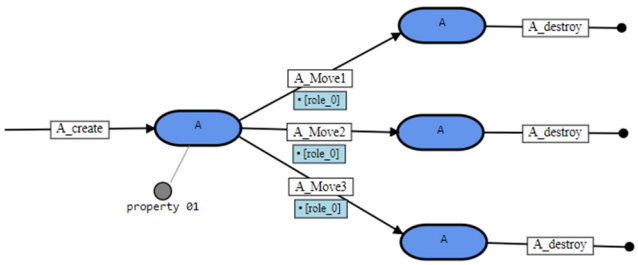


Fig. 16. Example of supply chain Graphical Representation (scGR): multiple operations.

observe that the declaration of the "asset_A_history" data structure in the smart contract in Fig. 17 includes the three default properties (lines 11–13), and the *property 01* one (line14). The procedure also creates a temporary nominal array which pairs each property with the functions where it appear as parameter. This data structure will be used by the next procedure to define the functions of the asset smart contract.

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. Since the supply chain described in the scGR in Fig. 14 performs the sequence of operations A_create, A_Move, and A_destroy, the smart contract in Fig. 17, includes four 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).

Instead, in the example of Fig. 16, the procedure would have found the operations: A_Move1, A_Move2, and A_Move3, besides A_create and A_destroy ones. For each function, a number of conditions are imposed by adding "require" statements to the code of the function itself. The first 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 right step of the production process represented by the scGR. For all the operations (with the exception of the sell one) the second condition, that is added by default as well, 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, the second condition 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. 17, 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 Fig. 17, for A_Move function, includes one authorization condition at line 28, because in the supply chain shown in Fig. 14 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.

The operations defined in the scJR are also used to build the"*web3js interface*" library. This interface is a collection of functions that can be used by the supply chain participants to invoke the smart contracts of the SCM system produced by our tool. In Fig. 20 an excerpt of the web3js interface library is shown where, for instance, the "A_create_obj" function invokes the "A_create" function of the smart contract representing the asset *A* (see Fig. 17), and the "f_A_Move" function invokes the "A_Move" function of the same smart contract. Please notice that the values of the variables in lines 1–4 must be inserted by the developer in the code of Fig. 20 after that the SCM system has been deployed on the blockchain.

8. Cost evaluation

In order to validate the *-chain framework as well as the blockchain based SCM systems generated using the *-chain tools, in this section we evaluate 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 network, 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.

The first set of results we present in this section, shown in Table 1, is related to the SCM system obtained from the supply chain represented in Fig. 14, varying the number of properties paired to asset *A*. Table 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.[14] For what concerns the deployment cost, as expected, Table 1 shows that it increases with 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 about

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

```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 {    infinite gas
        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 gas
        return store_A_s[_ID].A;
    }

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

Fig. 17. Example of solidity translation from the scJR.

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 lower than the deployment cost.

Table 1
Deployment costs and execution costs of the *A_create* and *A_Move* operations varying the number of properties of the asset *A*.

#Properties	#Move operations	Deployment cost (gas)	Execution cost (gas)
1	1	3 545 879	26 688 (A_create) 25 096 (A_Move)
2	1	3 585 972	27 753 (A_create) 25 106 (A_Move)
3	1	3 741 248	28 995 (A_create) 25 142 (A_Move)
5	1	4 045 538	31 372 (A_create) 25 140 (A_Move)
8	1	4 502 005	35 189 (A_create) 25 118 (A_Move)
10	1	4 781 756	42 103 (A_create) 25 154 (A_Move)

Table 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*.

#Properties	#Move operations	Deployment cost (gas)	Execution cost (gas)
1	1	3 545 879	26 688 (A_create) 25 096 (A_Move1)
1	2	3 920 498	26 503 (A_create) 25 118 (A_Move1)
1	3	4 295 137	26 559 (A_create) 25 140 (A_Move1)
1	4	4 669 597	26 585 (A_create) 25 170 (A_Move1)
1	5	5 044 301	26 550 (A_create) 25 196 (A_Move1)
3	3	4 664 381	28 947 (A_create) 25 130 (A_Move1)

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.

The second set of results, shown in Table 2, are obtained varying the number of operations defined in the supply chain for the asset *A*. For instance, Fig. 14 shows a supply chain where there is only one move operation defined on *A*, while Fig. 16 shows a supply chain where three distinct move operations are defined on *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 the table, which only refers to the moves operations. From the results reported in the first 5 rows of Table 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 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 1 for assets with 1 operation only.

Table 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 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 3 we observe that the trend is similar to the one shown in Table 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 the *A_Transform1* operation is not affected.

Table 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*.

#Properties	#Transform operations	Deployment cost (gas)	Execution cost (gas)
1	1	3 199 521	26 617 (A_create) 26 218 (A_Transform1)
1	2	3 253 698	26 615 (A_create) 26 218 (A_Transform1)
1	3	3 409 800	26 603 (A_create) 26 218 (A_Transform1)
1	4	3 565 987	26 693 (A_create) 26 218 (A_Transform1)
1	5	3 722 093	26 671 (A_create) 26 218 (A_Transform1)
3	3	3 837 147	28 914 (A_create) 26 218 (A_Transform1)

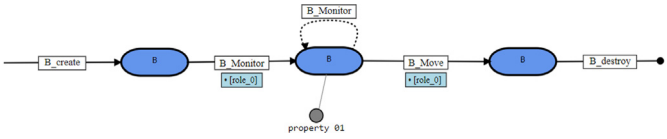


Fig. 18. Example of supply chain Graphical Representation (scGR): asset_monitor and asset_move operations.

We performed a forth set of experiments to evaluate the cost of the monitor operation exploiting the SCM system obtained from the supply chain represented in Fig. 18 and varying the number of properties that are involved in the monitor operation. The results are shown in Table 4. The deployment cost increases with the number of properties, and it ranges from about 3886K gas (1 property) to about 4454K gas (5 properties). We notice that the deployment costs of asset B shown in Table 4 are higher than the deployment costs of asset A shown in Table 1 (referring to the supply chain represented in Fig. 14). This is because the smart contract representing asset B is very similar to the smart contract representing asset A, but it embeds one operation more, the *B_Monitor*, and this affects the smart contract size. The cost of executing the *B_Monitor* operation slightly increases with the number of properties that are involved in the operation and, from 1 to 5 properties, it is about 25K gas.

Table 5 shows the results of a fifth set of experiments in which we measured the cost of the *sell* and *buy* operations,

Table 4
Deployment costs of asset B and execution costs of the *B_Monitor* operation varying the number of involved properties.

#Properties	Deployment cost (gas)	Execution cost (gas)
1	3 886 494	24 969
2	4 027 362	25 053
3	4 169 401	25 101
4	4 268 001	25 112
5	4 454 143	25 151

Table 5
Execution costs of sell and buy operations.

Operation	Execution cost (gas)
Sell	24 405
Buy	23 883

Table 6
Total deployment cost of a SCM system representing a sequential supply chain varying the number of assets.

#Assets	SCM system deployment cost (gas)
2	7 708 450
3	10 925 263
4	14 141 651

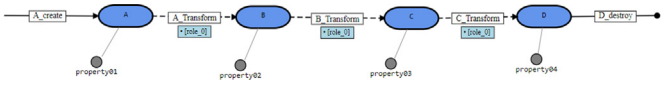


Fig. 19. Example of supply chain Graphical Representation (scGR): sequence of transform operations.

```
var LOCATION = '';
var ACCOUNT_ADDR ='';
var JSON_ABI = '';
var CONTRACT_ADDR = '';

var web3 = new Web3(new Web3.providers.HttpProvider(LOCATION));
web3.eth.defaultAccount = web3.eth.accounts[0];
var _yourAccount  = ACCOUNT_ADDR;
var abi = JSON_ABI;
var contract_addr = CONTRACT_ADDR;
// var contract_link = new web3.eth.Contract(abi, contract_addr);  // remove

function f_A_Move (the_ID) {
    let tempt = contract_link.methods.A_Move(the_ID)
    .send({ from: _yourAccount })
    .on('receipt', function(receipt){
        console.log(receipt);
    });
}

function A_create_obj (the_ID,  memory property_01) {
    let tempt = contract_link.methods.A_create(the_ID)
    .send({ from: _yourAccount })
    .on('receipt', function(receipt){
        console.log(receipt);
    });
}

function A_destroy_obj (the_ID,  memory property_01) {
    let tempt = contract_link.methods.A_destroy(the_ID)
    .send({ from: _yourAccount })
    .on('receipt', function(receipt){
        console.log(receipt);
    });
}

function A_read() {
    let temp = contract_link.methods.A_view()
        .call()
        .then(function(result){
            console.log(result);
        });
}
```

Fig. 20. Example of "web3js interface" library code.

used to transfer the ownership of an asset among two users. These operations have an execution cost of about 24K gas. In our experiments, we verified that such costs do not depend on the number of properties and operations that are defined on the asset.

Finally, the last set of experimental results evaluate the deployment cost of supply chains consisting of a sequence of n transform operations which, from an initial asset, produce a number of intermediate ones and then produces the final assets. Fig. 19 shows an example performing a sequence of 3 transform operations which involve 4 assets. Table 6 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 proportionally as the number of assets in the supply chain increases.

To complete the experimental evaluation of our approach, in the following we compare its costs with the ones of other SCM systems built on the Ethereum blockchain.

The authors of [22] propose a complete solution for blockchain based agrifood supply chains. In their solution they define a contract called *addTransaction* for tracing the execution of transactions on assets. From the results they show in the paper, we can see that tracing a transaction costs about 27K gas. The experiments were conducted on 2, 4, and 6 assets, and such cost does not increase significantly. The cost for one asset only is not given in the paper, but since such cost does not increase significantly varying the number of assets, we could assume that it would be around 27K gas as well. This cost is somehow comparable with the cost of performing an *asset_move* or an *asset_transform* operation on an asset in our framework (see Tables 1, 2, and 3) which ranges from about 25K to about 26K gas. Moreover, in our framework, we can execute an operation on multiple assets as in [22] by grouping a set of assets in a single container, and executing operations on such a container. Furthermore, in [22] the data describing an asset is actually stored off-chain (i.e., on IPFS[15]) and only the related hash is stored on-chain. Hence, although this solution guarantees the integrity of assets' data and allows to save gas, it does not guarantee that such data will be permanently kept available to the users of the SCM system.

Another blockchain based SCM system is presented in [23], and it concerns the oil supply chain tracing. In their system, the authors define a smart contract, called *CheckProgress* which provides a number of functions to keep track of some information about the oil during the production process. In particular, such smart contract provides three functions, called *CheckPressure*, *CheckTemperature*, and *CheckHumidity* which register on the blockchain the current value of pressure, temperature, and humidity, and the execution of such functions cost, respectively, about 30K gas, 14K gas, and 15K gas. The role of these functions in the SCM system could be somehow similar to the role of the *asset_monitor* function in our framework, which allows to register on the SCM system the current value of a property of an asset. The cost of the *CheckPressure* function (about 30K gas) is comparable to the cost of the *asset_monitor* of our framework (about 25K gas, see Table 4), while the other two functions, *CheckTemperature* and *CheckHumidity* cost considerably less of the *CheckPressure* function, and the authors of [23] do not describe the reason. However, we observe that one reason for the higher cost for the execution of *asset_monitor* function with respect to the *CheckTemperature* and *CheckHumidity* ones could be that the former also checks that the requesting user is the controller of the asset and holds the role required for executing the *asset_monitor* operation.

A third implementation of a blockchain based SCM system is described in [24]. This system is designed for the health care scenario and it is aimed at fighting drug counterfeiting, falsification,

15 https://ipfs.tech/.

and black market. In this work, the authors focus on the drug exchanges among the actors of the supply chain (manufacturers, wholesalers, distributors, pharmacies), while they do not take into account the drug production process. In particular, one of the smart contracts composing the system provides the function *purchaseMedicine*, which allows to track the transfer of the ownership of a given medicine from one stakeholder of the system to another. The experimental results presented in [24] show that the cost of executing the *purchaseMedicine* function goes from about 28K gas to about 30K gas. This cost is comparable with the cost of the *sell* function of our framework, shown in Table 5, which is about 24K gas. However, in our framework we require the execution of a second function, called *buy*, for finalizing the ownership transfer of an asset, which introduces an additional cost of about 24K gas. This function guarantees that the actor to whom the seller wants to transfer the ownership of the asset through the *sell* function actually accepts this transfer.

9. Related work

Several other works in literature analyze SCM systems, but most of them are focused on the management of a single use case, or, on best cases, they concern a specific SC topology. Surveys and other in-depth studies [25] analyze possible development, remaining however theoretical studies.

An example of the power of representative modeling for SC is described in [26]: here the authors develop a simulation model, using an object-oriented modeling framework to facilitate supply chain representation. Their approach is strictly process-centric, where the state or the maintenance of each asset is often not considered.

Authors in [27] have introduced blockchain-based food information security in SCM system. According to them, no solutions can achieve the traceability accuracy required for the real market: although a solution is provided, an implementation has not been developed. In this approach, there is no mention of product traceability but only to keep track of the finalized process. In a different way, our goal is to ensure product traceability: our system tracks both the production steps of an asset, and monitors its state, and its properties and its geophysical localization.

A similar study, presented in [28], 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 precomposed blocks but the users can define their low level or high level operations on the graphical interface. We exploited this idea, in order to build a model that can be translated into small conceptual building blocks, which can later be translated into functions within a smart contract.

Another approach, [29], involves the development of a contract modeling 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.

The Business Process Model and Notation [20] (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.

In [30], a model for smart contract generation based on the BPMN representation is shown. The proposed graphical DLS translates blockchain smart contracts using the DEMO modeling language [18]. This graphical representation makes it easier for the user to represent an operation workflow on the same asset.

In the BPMN choreography scenario in [31], 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.

A similar approach is presented in [17], 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 trace 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.

The comparative study presented in [32] analyzes the limits of the BPMN and Case Management Model and Notation (CMMN) approaches in the representation, schematization and possible translation into smart contracts. The authors highlight how in BPMN only specific types of processes can be easily 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.

In [16] 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.

In [33], 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.

The authors of [34] 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 supply chains. Instead, our system is

capable to track potentially each kind of supply goods, processes and workflow, included agricultural goods.

[35] 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.

In [36], 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.

The authors of [37] present an automatic smart contract generation framework exploiting ontologies and semantic rules to represent the domain specific knowledge, from which smart contracts templates are derived. Abstract syntax trees are then exploited to organize 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 supply chains through a user-friendly DSL implementing the general model we defined.

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 both earning an evaluation of constraints over the model and granting 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.

Instead of the classical product life-cycle model, [39] 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.

[40] 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.

[41] proposes the Gcoin project[16] 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.

[42] 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.

The authors of [22] propose a solution designed for the Agri-Food context, with a generic example. The paper describes the algorithms underlying the smart contracts building up the SCM system, and evaluates the expected costs in gas and USD. However, the paper does not show any example of smart contract implementing the presented algorithms. Unlike our implementation, in [22] each transaction involves a registration of the asset data on the Interplanetary File Storage System (IPFS), which is a secondary storage system, while only the hash of such data is recorded in the blockchain.

Similarly to the previous approach, authors in [43] 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.

[44] 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 [45] 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 [23], the authors propose a smart contract based oil supply chain management system, running on top of private and consortium blockchains. The proposed tracing system is based on a smart contract, called *CheckProgress* which is exploited to monitor and record on the blockchain the relevant information about the oil during the production process. The paper provides a table showing the execution costs in gas and USD of the functions defined by system smart contracts. Unlike our approach, besides being tailored for a specific scenario, this work is only focused on measuring some relevant parameters during the production steps.

The authors of [24] propose a solution for drug supply chain traceability. The proposed system aims at tracing the purchase and the delivery of drugs among the stakeholders of the healthcare scenario, in order to fight black market and drug counterfeiting. Hence, this system is not aimed to trace the drug production process. A test use case is proposed as a proof of concept, and the gas costs of the execution of the main functions of the smart contracts implementing the system are provided. As in previous cases, this system is not general since it is very specific for the auto retail industries.

The study in [46] presents an integration of the BCautoSCF framework, which deals with the management of the financing platform for the auto retail industry, from the gathering of the components to the sale. The proposed tool aims to automatically track the steps described by the BCautoSCF framework. The framework we developed is proposing the creation of three user interfaces, allowing the domain expert to manage the SC comprehensively: both in terms of monitoring or usage.

16 www.gcoin.com/.

10. Conclusion and future work

We presented *-chain, a framework defining a model, a DSL, and a set of tools for helping supply chain domain experts in designing and developing their supply chain Management Systems on top of blockchain technology. Our framework allows domain experts to easily represent the main types of supply chains through a simple and user-friendly graphical interface, and – once designed – to translate such supply chain scheme into: (i) a set of smart contracts skeletons implementing the related supply chain Management System, and (ii) two web interfaces for operating on such management system: one for the supply chain administrator, and the other for the supply chain participants. These interfaces allow supply chain administrators to register the participants to their Supply Chain Management systems assigning them roles and rights, and supply chain participants to the register the operations they executed on assets according to the designed supply chain and to the required roles.

We validated our framework, by using it to represent a well-known supply chain use case: the soybean traceability study case [7]. We built the corresponding model, adding more detail to the original schema, introducing role-based authorization controls and also other constraints on operations. We used our framework to translate the supply chain schema into solidity code, and to produce the supply chain Administrator and Participants interfaces.

We envisage a number of future works for enhancing the proposed framework. At first, we plan to better analyze the potential of the DSL, translating other specific use cases for agrifood supply chains: Protected Designation of Origin (PDO[17]) olive oil and beef steak. 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 GDPR-compliant identity management, instead of outsourcing the storage of sensitive data to off-chain systems.

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.

CRediT authorship contribution statement

Stefano Bistarelli: Conceptualization, Methodology, Writing – review & editing, Investigation, Supervision, Validation. **Francesco Faloci:** Software, Data curation, Writing – original draft, Visualization, Investigation. **Paolo Mori:** Conceptualization, Methodology, Writing – original draft, Writing – review & editing, Investigation, Supervision, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The authors do not have permission to share data.

[17] https://ec.europa.eu/info/food-farming-fisheries/food-safety-and-quality/certification/quality-labels/quality-schemes-explained_en#pdo.

Acknowledgments

Stefano Bistarelli and Francesco Faloci are member of the INdAM Research group GNCS and of Consorzio CINI. This work was partially supported by: project "SERICS" (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU; project "FICO, funded by Ricerca di Base 2021", University of Perugia, Italy; project "BLOCKCHAIN4FOODCHAIN, funded by Ricerca di Base 2020", University of Perugia, Italy; GNCS-INdAM project CUP E53C22001930001; Project "Vitality", CUP J97G2000170005;

Appendix. List of acronyms

Acron.	Full name	Sec
SC	Supply Chain	1
SCM system	Supply Chain Management system	1
DSL	Domain Specific graphical Language	1
BT	Blockchain Technology	1
DLT	Distributed Ledger Technology	2.1
scGM	supply chain General Model	3
scDI	supply chain Design Interface	3
scGR	supply chain Graphical Representation	3
scJR	supply chain JSON Representation	3
scMT	supply chain Model Translator	3
scMIB	supply chain Managing Interfaces Builder	3
scAI	supply chain Administration Interface	3
scPI	supply chain Participant Interface	3
RBAC	Role-Based Access Control	4
SBP	Soy Beans Producer	5
BPMN	Business Process Model and Notation	9
CMMN	Case Management Model and Notation	9

References

[1] H. Stadtler, C. Kilger, Supply Chain Management and Advanced Planning: Concepts, Models, Software, and Case Studies, fourth ed., Springer Publishing Company, Incorporated, 2008, http://dx.doi.org/10.1007/978-3-540-74512-9.

[2] A. Litke, D. Anagnostopoulos, T. Varvarigou, Blockchains for supply chain management: Architectural elements and challenges towards a global scale deployment, Logistics 3 (2019) 5, http://dx.doi.org/10.3390/logistics3010005.

[3] P. Dutta, T.-M. Choi, S. Somani, R. Butala, Blockchain technology in supply chain operations: Applications, challenges and research opportunities, Transp. Res. E 142 (2020) 102067, http://dx.doi.org/10.1016/j.tre.2020.102067, URL: https://www.sciencedirect.com/science/article/pii/S1366554520307183.

[4] J. Solarte-Rivera, A. Vidal-Zemanate, C. Cobos, J.A. Chamorro-Lopez, T. Velasco, Document management system based on a private blockchain for the support of the judicial embargoes process in Colombia, in: R. Matulevicius, R.M. Dijkman (Eds.), Advanced Information Systems Engineering Workshops - CAiSE 2018 International Workshops, Tallinn, Estonia, June 11-15, 2018, Proceedings, in: Lecture Notes in Business Information Processing, vol. 316, Springer, 2018, pp. 126–137, http://dx.doi.org/10.1007/978-3-319-92898-2_10.

[5] E. Martiri, G. Muca, DMS-XT: A blockchain-based document management system for secure and intelligent archival, in: E. Xhina, K. Hoxha (Eds.), 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, in: CEUR Workshop Proceedings, vol. 2280, CEUR-WS.org, 2018, pp. 70–74, URL: http://ceur-ws.org/Vol-2280/paper-10.pdf.

[6] R. Azzi, R.K. Chamoun, M. Sokhn, The power of a blockchain-based supply chain, Comput. Ind. Eng. 135 (2019) 582–592, http://dx.doi.org/10.1016/j.cie.2019.06.042.

[7] K. Salah, N. Nizamuddin, R. Jayaraman, M. Omar, Blockchain-based Soybean traceability in agricultural supply chain, IEEE Access 7 (2019) 73295–73305, http://dx.doi.org/10.1109/ACCESS.2019.2918000.

[8] S. Bistarelli, F. Faloci, P. Mori, Towards a graphical DSL for tracing supply chains on blockchain, in: R. Chaves, D.B. Heras, A. Ilic, D. Unat, R.M. Badia, A. Bracciali, P. Diehl, A. Dubey, O. Sangyoon, S.L. Scott, L. Ricci (Eds.), Euro-Par 2021: Parallel Processing Workshops - Euro-Par 2021 International Workshops, Lisbon, Portugal, August 30-31, 2021, Revised Selected Papers, in: Lecture Notes in Computer Science, 13098, Springer, 2021, pp. 219–229, http://dx.doi.org/10.1007/978-3-031-06156-1_18, https://doi.org/10.1007/978-3-031-06156-1_18.

[9] S. Bistarelli, F. Faloci, 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, IEEE, 2021, pp. 164–171, http://dx.doi.org/10.1109/BCCA53669.2021.9656987.

[10] S. Malik, S.S. Kanhere, R. 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, IEEE, 2018, pp. 1–10, http://dx.doi.org/10.1109/NCA.2018.8548322.

[11] H. Al-Breiki, M.H.U. Rehman, K. Salah, D. Svetinovic, Trustworthy blockchain oracles: Review, comparison, and open research challenges, IEEE Access 8 (2020) 85675–85685, http://dx.doi.org/10.1109/ACCESS.2020.2992698.

[12] N. Trautmann, C. Fündeling, Supply chain management and advanced planning in the process industries, in: K. Waldmann, U.M. Stocker (Eds.), 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, 2006, pp. 503–508, http://dx.doi.org/10.1007/978-3-540-69995-8_80.

[13] J. Swaminathan, S. Smith, N. Sadeh, Modeling supply chain dynamics: A multiagent approach, Decis. Sci. 29 (2000) http://dx.doi.org/10.1111/j.1540-5915.1998.tb01356.x.

[14] K. Govindan, H. Soleimani, D. Kannan, Reverse logistics and closed loop supply chain: A comprehensive review to explore the future, European J. Oper. Res. 240 (3) (2015) 603–626, http://dx.doi.org/10.1016/j.ejor.2014.07.012.

[15] A. Tetteh, Q. Xu, G. Sun, Supply chain distribution networks: Single-, dual-, & omni-channel, in: Interdisciplinary Journal of Research in Business, Vol. 3, 2014, pp. 63–73.

[16] E. Lhassan, R. Ali, F. 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), 2018, pp. 1–10, http://dx.doi.org/10.1109/GOL.2018.8378078.

[17] F. Corradini, A. Marcelletti, A. Morichetta, A. Polini, B. Re, F. Tiezzi, ChorChain: A model-driven framework for choreography-based systems using blockchain, in: A. Marrella, D.T. Dupré (Eds.), 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, in: CEUR Workshop Proceedings, vol. 2952, CEUR-WS.org, 2021, pp. 26–32, URL: http://ceur-ws.org/Vol-2952/paper_294a.pdf.

[18] J.L. Dietz, DEMO: Towards a discipline of organisation engineering, European J. Oper. Res. 128 (2) (2001) 351–363, http://dx.doi.org/10.1016/S0377-2217(00)00077-1, URL: https://www.sciencedirect.com/science/article/pii/S0377221700000771, Complex Societal Problems.

[19] A. Baouab, W. Fdhila, O. Perrin, C. Godart, Towards decentralized monitoring of supply chains, in: 2012 IEEE 19th International Conference on Web Services, 2012, pp. 600–607, http://dx.doi.org/10.1109/ICWS.2012.13.

[20] M. von Rosing, S. White, F. Cummins, H. de Man, Business process model and notation - BPMN, in: M. von Rosing, H. von Scheel, A. Scheer (Eds.), The Complete Business Process Handbook: Body of Knowledge from Process Modeling To BPM, Vol. I, Morgan Kaufmann/Elsevier, 2015, pp. 429–453, http://dx.doi.org/10.1016/B978-0-12-799959-3.00021-5.

[21] D.F. Ferraiolo, D.R. Kuhn, R. Chandramouli, Role-Based Access Control, second ed., Artech House, Inc., USA, 2007.

[22] A. Shahid, A. Almogren, N. Javaid, F.A. Al-Zahrani, M. Zuair, M. Alam, Blockchain-based agri-food supply chain: A complete solution, IEEE Access 8 (2020) 69230–69243, http://dx.doi.org/10.1109/ACCESS.2020.2986257.

[23] B. Haque, R. Hasan, O.M. Zihad, SmartOil: Blockchain and smart contract-based oil supply chain management, IET Blockchain 1 (2–4) (2021) 95–104, http://dx.doi.org/10.1049/blc2.12005.

[24] K.C. Bandhu, R. Litoriya, M. Lowanshi, L. Chouhan, S. Jain, Making drug supply chain secure traceable and efficient: a Blockchain and smart contract based implementation, Multimedia Tools Appl. (2022) http://dx.doi.org/10.1007/s11042-022-14238-4.

[25] M. Pournader, Blockchain applications in supply chains, transport and logistics: a systematic review of the literature, Int. J. Prod. Res. 58 (2019) http://dx.doi.org/10.1080/00207543.2019.1650976.

[26] D. van der Zee, J.G.A.J. van der Vorst, A modeling framework for supply chain simulation: Opportunities for improved decision making, Decis. Sci. 36 (1) (2005) 65–95, http://dx.doi.org/10.1111/j.1540-5915.2005.00066.x.

[27] M. Nakasumi, Information sharing for supply chain management based on block chain technology, in: 2017 IEEE 19th Conference on Business Informatics (CBI), Vol. 01, 2017, pp. 140–149, http://dx.doi.org/10.1109/CBI.2017.56.

[28] D. Mao, F. Wang, Y. Wang, Z. Hao, Visual and user-defined smart contract designing system based on automatic coding, IEEE Access 7 (2019) 73131–73143, http://dx.doi.org/10.1109/ACCESS.2019.2920776.

[29] M. Wöhrer, U. Zdun, Domain specific language for smart contract development, in: 2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), 2020, pp. 1–9, http://dx.doi.org/10.1109/ICBC48266.2020.9169399.

[30] M. Skotnica, J. Klicpera, R. Pergl, Towards model-driven smart contract systems - code generation and improving expressivity of smart contract modeling, in: G. Sergio, S. Pedro, A. David, G. Giancarlo, P. Robert, P. Henderik A. (Eds.), CIAO! Doctoral Consortium, EEWC Forum 2020, in: CEUR Workshop Proceedings, vol. 2825, Bolzano, Italy, 2021, pp. 1–16, URL: http://ceur-ws.org/Vol-2825/.

[31] L. Spalazzi, F. Spegni, A. Corneli, B. Naticchia, Blockchain based choreographies: The construction industry case study, Concurr. Comput.: Pract. Exper. n/a (n/a) (2021) e6740, http://dx.doi.org/10.1002/cpe.6740.

[32] F. Milani, L. García-Bañuelos, S. Filipova, M. Markovska, Modelling blockchain-based business processes: a comparative analysis of BPMN vs CMMN, Bus. Process. Manage. J. 27 (2) (2021) 638–657, http://dx.doi.org/10.1108/BPMJ-06-2020-0263.

[33] D. Tse, B. Zhang, Y. Yang, C. Cheng, H. Mu, Blockchain application in food supply information security, in: 2017 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), 2017, pp. 1357–1361, http://dx.doi.org/10.1109/IEEM.2017.8290114.

[34] K. Biswas, V. Muthukkumarasamy, W. Lum, Blockchain based wine supply chain traceability system, in: Future Technologies Conference (FTC) 2017, Vancouver, BC, Canada, 2017.

[35] F. 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, 2017, pp. 1–6, http://dx.doi.org/10.1109/ICSSSM.2017.7996119.

[36] M.P. Caro, M.S. Ali, M. Vecchio, R. Giaffreda, Blockchain-based traceability in agri-food supply chain management: A practical implementation, in: 2018 IoT Vertical and Topical Summit on Agriculture - Tuscany (IOT Tuscany), 2018, pp. 1–4.

[37] O. Choudhury, N. Rudolph, I. Sylla, N. Fairoza, A. Das, Auto-generation of smart contracts from domain-specific ontologies and semantic rules, in: 2018 IEEE International Conference on Internet of Things (IThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 2018, pp. 963–970, http://dx.doi.org/10.1109/Cybermatics_2018.2018.00183.

[38] D. Remenska, J. Templon, T.A.C. Willemse, P. Homburg, K. Verstoep, A.C. Ramo, H.E. Bal, From UML to process algebra and back: An automated approach to model-checking software design artifacts of concurrent systems, in: G. Brat, N. Rungta, A. Venet (Eds.), NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings, in: Lecture Notes in Computer Science, vol. 7871, Springer, 2013, pp. 244–260, http://dx.doi.org/10.1007/978-3-642-38088-4_17.

[39] M. Westerkamp, F. Victor, A. Küpper, Blockchain-based supply chain traceability: Token recipes model manufacturing processes, in: 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), IThings/GreenCom/CPSCom/SmartData 2018, Halifax, NS, Canada, July 30 - August 3, 2018, IEEE, 2018, pp. 1595–1602, http://dx.doi.org/10.1109/Cybermatics_2018.2018.00267.

[40] M. Holland, J. Stjepandic, C. 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, IEEE, 2018, pp. 1–8, http://dx.doi.org/10.1109/ICE.2018.8436315.

[41] J.-H. Tseng, Y.-C. Liao, B. Chong, S.-w. Liao, Governance on the drug supply chain via gcoin blockchain, Int. J. Environ. Res. Public Health 15 (2018) 1055, http://dx.doi.org/10.3390/ijerph15061055.

[42] T. Peltoniemi, J. Ihalainen, Evaluating blockchain for the governance of the plasma derivatives supply chain: How distributed ledger technology can mitigate plasma supply chain risks, Blockchain Healthc. Today 2 (2019) http://dx.doi.org/10.30953/bhty.v2.107.

[43] J. Hao, Y. Sun, H. Luo, A safe and efficient storage scheme based on blockchain and IPFs for agricultural products tracking, J. Comput. (Taiwan) 29 (2018) 158–167, http://dx.doi.org/10.3966/199115992018122906015.

[44] S. Wang, Y. Zhang, Y. Zhang, A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems, IEEE Access 6 (2018) 38437–38450, http://dx.doi.org/10.1109/ACCESS.2018.2851611.
[45] Z. Li, H. Wu, B. King, Z. Ben Miled, J. Wassick, J. Tazelaar, A hybrid blockchain ledger for supply chain visibility, in: 2018 17th International Symposium on Parallel and Distributed Computing (ISPDC), 2018, pp. 118–125, http://dx.doi.org/10.1109/ISPDC2018.2018.00025.
[46] J. Chen, T. Cai, W. He, L. Chen, G. Zhao, W. Zou, L. Guo, A blockchain-driven supply chain finance application for auto retail industry, Entropy 22 (2020) 95, http://dx.doi.org/10.3390/e22010095.

Further reading

[1] R. Chaves, D.B. Heras, A. Ilic, D. Unat, R.M. Badia, A. Bracciali, P. Diehl, A. Dubey, O. Sangyoon, S.L. Scott, L. Ricci (Eds.), Euro-par 2021: parallel processing workshops - euro-par 2021 international workshops, lisbon, portugal, august 30-31, 2021, revised selected papers, in: Lecture Notes in Computer Science, 13098, Springer, 2022, http://dx.doi.org/10.1007/978-3-031-06156-1, https://doi.org/10.1007/978-3-031-06156-1.

Stefano Bistarelli Full Professor at University of Perugia, is an active researcher in the field of Knowledge Representation and Reasoning, with a particular interest in Argumentation, Constraint Programming, Cybersecurity and blockchain. He received his Phd in Computer Science in Pisa, awarded by both the Italian section of the European Association for Theoretical Computer Science (EATCS) and by the Italian Association for Artificial Intelligence (AI*IA). Later, in 2004, an extended version of his doctoral work has been published as a volume in the Springer Jacketed LNCS series. This book represents at today an important reference for those who are deeply involved in the (Soft) Constraint Programming area. Currently he is leading the Italian DLT working group collecting many of the italian scientists working on blockchain. He is an active researcher counting more than 300 papers and 4000 citations on google scholar.

Francesco Faloci is a Ph.D. student at University of Camerino supported by Istituto di Informatica e Telematica of Consiglio Nazionale delle Ricerche in Pisa and University of Perugia, under the supervision of Professor Stefano Bistarelli and Dr. Paolo Mori. His main research interests are in the field of Knowledge Representation and Reasoning, Cybersecurity and blockchain technologies.

Paolo Mori is a senior researcher at Istituto di Informatica e Telematica of Consiglio Nazionale delle Ricerche. He received his Ph.D. from the University of Pisa in 2003. His research interests involve trust, security and privacy in distributed environments, focusing on access/usage control, data privacy in (Distributed) Online Social Networks, and Blockchain technology and its applications. He was Program Co-Chair of the 2nd–8th editions of the International Conference of Information System Security and Privacy (ICISSP 2016–2022). He is (co-)author of 100+ papers published on international journals and conference/workshop proceedings. He is usually actively involved in European and Italian research projects on information and communication security.