Decentralized Enforcement of DEMO Action Rules Using Blockchain Smart Contracts

Marta Aparício[12] and Sérgio Guerreiro[12] Pedro Sousa[123]

[1] Instituto Superior Técnico, University of Lisbon,
Av. Rovisco Pais 1, 1049-001 Lisbon, Portugal
[2] INESC-ID, Rua Alves Redol 9, 1000-029 Lisbon, Portugal
[3] Link Consulting SA, Av. Duque de Avila 23, 1000-138 Lisbon, Portugal
{marta.m.s.aparicio, sergio.guerreiro, pedro.manuel.sousa}@tecnico.ulisboa.pt

Abstract. Blockchain technology is a solution to coordinate inter-organizational processes involving untrusted parties. Blockchain is by design an immutable record, so it is non-trivial or even infeasible to update Smart Contract. Concerning the automatic generation of Smart Contracts, Model-Driven Engineering is a software engineering method that uses models with various views and levels of abstraction to achieve different goals in the software development process. Models with a lower level of abstraction can be used to directly generate software production code. This paper sets out to address the hypothesis of using Blockchain Smart Contracts to implement DEMO Action Models. If the hypothesis ends up being admissible, through demonstration and verification, this will mean the reuse of Ontological models in line with the correct implementation of the Smart Contract on Blockchain.

Key words: Blockchain, DEMO, DEMO Action Model, Ethereum, Smart Contract

1 Introduction

As enterprises are complex systems involving both human and technological aspects and are highly influenced by the environment in which they operate, architecting an enterprise and the information systems that support their functioning is not an easy task [1].

Each enterprise has to manage several processes. Business Processes (BPs) are what enterprises do whenever they deliver a service or a product to customers. Dumas et al. [2] defines a BP as *"a collection of inter-related events, activities, and decision points that involve a number of actors and objects, which collectively lead to an outcome that is of value to at least one customer"*. Armed with this definition of a BP, Business Process Management (BPM), was also defined in [2], as a *"body of methods, techniques, and tools to identify, discover, analyze, redesign, execute, and monitor business processes to optimize their performance"*.

Information Technologies (IT) plays a significant role in BPs or processes in general, as more and more of them are supported by IT systems. Business Pro-

cess Management Systems (BPMSs) are just one type of IT tool that supports the implementation and execution of BPs. There are many others, including Enterprise Resource Planning (ERP) systems, Customer Relationship Management (CRM) systems, and Document Management Systems (DMSs). BPMSs are mainly focused on the automation of BPs within a given organization (intra-organizational processes). Many processes, however, span across organizational boundaries (inter-organizational processes).

The continuous development of digitization and internationalization in various fields such as e-commerce or supply chain management [3] has led to an increasing shift to more collaboration in BPs. A characteristic of collaborative BPs is that they are composed of different sub-processes executed by various organizations. By their nature, sub-processes carried out by one organization are usually beyond the domain of influence of all other collaborators [4]. But in the context of the process, the results of the sub-process may be important to other collaborators. Therefore, from the perspective of all other collaborators, this creates uncertainty in the process. Where there is uncertainty, trust is needed [5]. Therefore, collaborative BPs are trust-intensive.

The intricate environment created by global trading leads to enforcing trust by centralized organizations such as insurances and banks, which themselves are supported by the ultimate centralized authority in the system, the state institutions. Nonetheless, this type of organization forces business partners to depend upon third parties to manage and enforce those contractual agreements.

BC technology can be seen as a solution to coordinate inter-organizational processes involving untrusted parties [2]. This technology provides a way to record something that happened to ensure that it cannot be deleted once recorded. Smart Contracts are mechanisms, that exist in latter BC generations, that ensure that a given routine is executed every time a transaction, of a given type, is recorded. Traditionally, the coordination of all parties in a transaction is accomplished through message exchange. Instead of exchanging messages, all parties involved in the inter-organizational process can execute transactions on the BC. This alternative method ensures that important business rules are always followed.

Through a distributed ledger and shared business logic in the form of Smart Contracts, enterprises can execute shared business logic and monitor the state of process instances [4]. Compared to other enterprise integration efforts, such as Service-Oriented Architectures (SOAs), BC allows organizations to use a single shared database to manage and operate their shared BP.

Although the feasibility of implementing BPs on top of BC has been proven, BC is considered a relatively new technology compared to others. As Gartner claims [6], BC technology is still very immature to support most of the potential use cases and there's still a tremendous amount of research, implementation, and adoption to be done. Further, building systems on BC is non-trivial due to the steep learning curve of the BC technology. According to another survey by Gartner [7], *"23 percent of [relevant surveyed] CIOs said that blockchain requires*

the most new skills to implement of any technology area, while 18 percent said that blockchain skills are the most difficult to find" [7].

A concrete and real example of a problem that occurred in a BC was analyzed and discussed by Alex Norta in [8]. The paper refers to a crowdfunding project that was hacked due to security flaws, which resulted in a loss of $50 million. Concerning this problem, North states *"The incident shows it is not enough to merely equip the protocol layer on top of a BC with a Turing-complete language such as Solidity to realize smart-contract management. Instead, we propose in this keynote paper that is crucial to address a gap for secure smart-contract management pertaining to the currently ignored application-layer development"* [8]. This suggests that the automatic generation of Smart Contracts would have the added value of creating a new level of security.

Concerning the automatic generation of Smart Contracts, several approaches can be followed, one of which being a Model-Driven Engineering (MDE) approach. MDE is a software engineering method that uses models with various views and levels of abstraction to achieve different goals in the software development process. Models with a lower level of abstraction can be used to directly generate software production code. In the context of BC applications, MDE is of particular importance as the BC is by design an immutable record, so it is non-trivial or even infeasible to update Smart Contracts [9].

In an industrial environment, BPs, as part of enterprises, can be decomposed into two parts: the BP model, the desired behavior of a BP; and the BP instance, the actual behavior in operation in a specific enterprise. An ontology model of a system is fully independent of the implementation, and it only shows the essential features. A good example of an ontology is the Design & Engineering Methodology for Organizations (DEMO) method of Enterprise Ontology (EO) because it treats the enterprise as a system [10].

DEMO has been widely accepted in both scientific research and practical appliances [11, 12]. In DEMO, an enterprise is seen as a system of people and their relations, authority and responsibility. The usage of a strongly simplified models that focus on people forms the basis of DEMO. By using a language that is common in the enterprise, the understanding of such models is guaranteed, even though they're abstract and have a conceptual nature.

Compared with its implementation model, the EO model reduces complexity [13]. This reduction in complexity makes the organization more transparent. It also shows the consistency between all areas within the enterprise, such as BPs and workflows.

This paper, as the title suggests, intends to explore the implementation of DEMO Action Models in BC Smart Contracts. To demonstrate and later discuss the ontological implications of this implementation, a Rent-A-Car use case was created. A BC Smart Contract was generated and a simulation of the functionality was performed as well.

2 Related Research

There have been attempts at raising the level of abstraction from code-centric to model-centric SCs development. The different approaches tried so far, can be divided into three: the Agent-Based Approach, the State Machine Approach, and the Process-based approach [16].

The Agent-Based Approach described by [17] proposes a modeling approach that supports the semi-automated translation of human-readable contract representations into computational equivalents to enable the codification of laws into verifiable and enforceable computational structures that reside within a public BC. They identify SC components that correspond to real-world institutions and propose a mapping using a domain-specific language to support the contract modeling process. The concept Grammar of Institutions [18] is used to decompose institutions into rule-based statements. These statements are then compiled in a structured formalization. In this case, the statements are constructed from five components, abbreviated to ADICO. The Attributes describing an actor's characteristics or attributes. The Deontic describing the nature of the statement as an obligation, permission, or prohibition. The AIm describing the action or outcome that this statement regulates. The Conditions describing the contextual conditions under which this statement holds. And the Or else describing consequences associated with non-conformance. Using these components, statements on the execution of the SC are made.

The State Machine Approach is based on the observation that SCs act as state machines. A SC is in an initial state and a transaction transitions the contract from one state to the next. The possibility of SCs as state machines is also described in the Solidity specification. [19] show that the transformation of the Finite State Machine to Solidity is partly automated since to ensure Solidity code quality, some manual coding might be necessary or added through plugins.

In [20] was proposed as a tool to support inter-organizational processes through BC technology. To ensure that the joint process is correctly executed, the control flow and business logic of inter-organizational business processes are compiled from the processes models into smart contracts. Weber et al. developed a technique to integrate BC into the choreography of processes to maintain trust, employing triggers and web services. By storing the status of process execution across all involved participants, as well as coordinating the collaborative business process execution in the BC. The validation was made against the ability to distinguish between conforming and non-conforming traces. [21] presented an optimization in regards to the already presented paper [20]. To compile BPMN models into a Smart Contract in Solidity Language, the BPMN model is first translated into a reduced Petri Net. Only after this first step, the reduced Petri is compiled into a Solidity Smart Contract. Compared to [20], [21] managed to decrease the amount of paid resources and achieve higher throughput. Caterpillar, first presented in [22] and further discussed in [23], is an open-source Business Process Management System that runs on top of the Ethereum BC. Like any BPMS, Caterpillar supports the creation of instances of a process model (captured in BPMN) and allows users to track the state of process instances and

to execute tasks thereof. The specificity of Caterpillar is that the state of each process instance is maintained on the Ethereum BC, and the workflow routing is performed by Smart Contracts generated by a BPMN-to-Solidity compiler. Given a BPMN model (in standard XML format), it generates a Smart Contract (in Solidity), which encapsulates the workflow routing logic of the process model. Specifically, the Smart Contract contains variables to encode the state of a process instance, and scripts to update this state when-ever a task completes or an event occurs. Tran et al. [24] presented a tool that automatically creates a well-tested Smart Contract code from specifications that are encoded in the business process and data registry models based on the implemented model transformations. The BPMN translator can automatically generate Smart Contracts in Solidity from BPMNmodels while the registry generator creates Solidity Smart Contract based on the registry models. The BPMN translator takes an existing BPMN business process model as input and outputs a Smart Contract. This output includes the information to call registry functions and to instantiate and execute the process model. The registry generator takes data structure information and registry type as fields, and basic and advanced operations as methods, from which it generates the registry Smart Contract. This work builds upon already seen works, such as [21, 20], for the BPMN translation algorithms.

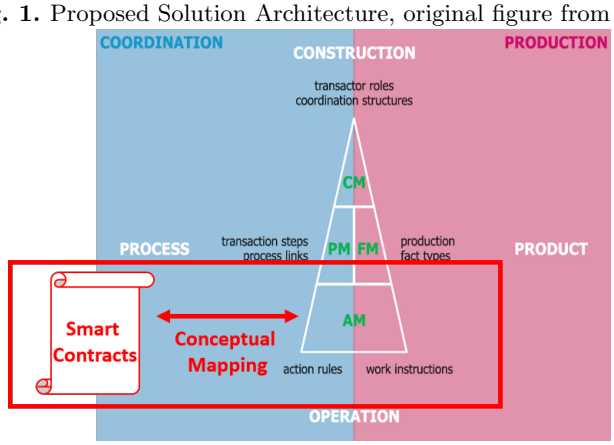
3 Methodologies Used

3.1 DEMO

Dietz uses the $\psi - theory$ to construct a methodology providing an ontological model of an organization, i.e. a model that is coherent, comprehensive, consistent, and concise, and that only shows the essence of the operation of an organization model. This methodology is called Design and Engineering Methodology for Organizations (DEMO). DEMO has been widely accepted in both scientific research and practical appliance [11]. In DEMO, an enterprise is seen as a system of people and their relations, authority and responsibility. The usage of a strongly simplified models that focus on people forms the basis of DEMO. By using a language that is common in the enterprise, the understanding of such models is guaranteed, even though they're abstract and have a conceptual nature. The core concept of DEMO is a transaction, fully based on the $\psi - theory$.

This paper starts with a belief that a DEMO transaction is represented as a contract in BC. The contract has its own address, internal storage, attributes, methods and it is callable by either an external actor or another contract. This is the functionality needed to represent a DEMO transaction. They implemented the execution of DEMO transactions according to the DEMO Machine [31] and associated theories. Now, this present work argues that the SCs automated generation could be done directly from the Action Model. It believes that the structure and content of a SC can be directly mapped to each action rule. Since that, by creating the action rules it is also being created the logic on which the SCs operate. In fact, the action rules contain all the decomposed detail of the above

Fig. 1. Proposed Solution Architecture, original figure from [30]



models, the basis of the DEMO methodology is exactly the Action Model, as can be seen in Figure 3.1. The Construction Model specifies the construction of the organization, specifies the identified transaction types and the associated actor roles, as well as the information links between the actor roles and the information banks. By occupying the top of the triangle it is suggested that is the most concise model. The Process Model contains, for every transaction type in the Construction Model, the specific transaction pattern of the transaction type. And, also contains the causal and conditional relationships between transactions. The Process Model is put just below the Construction Model in the triangle because it is the first level of detailing of the Construction Model, namely, the detailing of the identified trans-action types. The Action Model specifies the action rules that serve as guidelines for the actors in dealing with their agenda. The Action Model is put just below the Process Model in the triangle because it is the second level of detailing of the Construction Model, namely, the detailing of the identified steps in the Process Model of the transaction types in the Construction Model. At the ontological level of abstraction, there is nothing below the Action Model. The Fact Model is put on top of the Action Model in Figure 3.1 because it is directly based on the Action Model; it specifies all object classes, fact types, and ontological coexistence rules that are contained in the Action Model. The Action Model is in a very literal sense the basis of the other aspect models since it contains all information that is (also) contained in the Construction Model, Process Model, and Fact Model; but differently. These models have as if a zoom in (Action Model) zoom out (Construction Model) relationship between each other. The Action Model is the most detailed and comprehensive aspect model.

3.2 Model-Driven Engineering

MDE is a software engineering method that uses models with various views and levels of abstraction to achieve different goals in the software development

process. Models with a lower level of abstraction can be used to directly generate software production code.

In the context of BC applications, MDE is of particular importance as the BC is by design an immutable record, so it is non-trivial or even infeasible to update Smart Contracts [9].

MDE tools can generate well-tested code implementing best practices and help developers manage software complexity by only focusing on building high-level models without requiring expert development knowledge [27], thereby reducing the occurrence of vulnerable code that may easily be attacked. With research on BC technology on the rise and new ways of applying this technology to different domains being discovered [28], it becomes necessary not to overfit to a specific BC platform. Model abstraction can avoid this problem, and Model-Driven development tools can be applied at multiple BC platforms. Models are easier to understand than code, which improves development efficiency [29]. It is easier to check the correctness of a model, and Model-Driven tools can ensure that the deployed code is not modified after it is generated from the model. MDE development of BC applications can facilitate communication with domain experts [24], who can examine models to understand how their ideas are represented in the system.

3.3 Blockchain

Nakamoto [25] described BC as an architecture that gives participants the ability to perform electronic transactions without relying on trust. What makes this possible is that each block contains some data, the hash of the block, and the hash of the previous block. The data that is stored inside a block depends on the type of BC, but normally stores the details of multiple transactions, each with identification for the sender, the receiver, and the asset. A block also has a hash that identifies its content and it's always unique. If something is changed inside a block, that would cause the hash to change. That's why hashes are very useful to detect changes in blocks. The hash of the previous block effectively creates a chain of blocks and it's this technique that makes a BC so secure.

However, the hashing technique is not enough, with the high computational capacity that exists today, where a computer can calculate hundreds of thousands of hashes, per second [16]). To mitigate this problem, BC has a consensus mechanism called Proof-of-Work. This mechanism slows down the creation of new blocks since if a block is tempered the Proof-of-Work of all the previous blocks has to be recalculated. So, the security of BC comes from its creative use of hashing and a Proof-of-Work mechanism. As long as a majority of CPU power is controlled by nodes that are not cooperating to attack the network, they'll outpace attackers. Of course, its distributive nature also adds a level of security, since instead of using a central entity to manage the chain, BC uses a peer-to-peer network where anyone can join. Information is broadcast on a best effort basis, and nodes can leave and rejoin the network at will, accepting the longest Proof-of-Work chain as proof of what happened while they were gone.

3.4 Smart Contracts

In the context of BC, in particular second-generation BC, SCs are just like contracts in the real world. The only difference is that they are completely digital, in the sense that they are both defined by software code and executed or enforced by the code itself automatically without discretion. The trust issue is also addressed, once Smart Contracts are stored on a BC, they inherit some interesting properties: immutable and distributed. Being immutable means that once a SC is created, it can never be changed again. So, it is 't possible to tamper with the code of the contract. Being distributed means that the output of the contract is validated by every node on the network. So, a single node can not force the behavior of the contract since it is dependent on the other nodes. Like all algorithms, SCs may require input values and only act if certain pre-defined conditions are met. When a particular value is reached the SC changes its state and executes the functions, that are programmatically predefined algorithms, automatically triggering an event on the BC. If false data is inputted into the system, then false results will be output [26].

4 Our Approach

Solidity is a high-level programming language to implement SCs specially design for the Ethereum Virtual Machine (EVM). Solidity was chosen because itis developed under Ethereum and it is the most used language for SCs for EVM. The building block in Solidity is a contract that is similar to a class in object-oriented programming. A Contract contains persistent data in state variables, functions to operate on this data and it also supports inheritance. A Contract can further contain function modifiers, events, struct types, and other structures to allow the implementation of complex contracts, and full usage of EVM and BC capabilities. A SC written in Solidity can be created either through an Ethereum transaction or by another already running contract, just like an instance of a class would be created. Either way, the contract code is then compiled to the EVM bytecode, a new transaction is created holding the code and deployed to BC, returning the address of the contract for further interaction.

Contracts often act as a state machine, which means that they have certain stages in which they behave differently or in which different functions can be called. A function call often ends a stage and transitions the contract into the next stage, this is known as the State Machine common pattern [32]. Through the DEMO theory, it is known that actors interact by means of creating and dealing with C-facts. Since these contracts will model interactions it seems fit to model C-facts into stages, these stages are implemented as Enums. Enums are a way to create a user-defined type in Solidity, for this particular application seven stages, corresponding to the coordination facts: Initial; Requested; Promised; Declined; Declared; Accepted; Rejected. The Initial stage was created with the assumption that the deployment of a contract by someone doesn't mean they want to immediately start the transactions. Function Modifiers can automatically check a condition before executing a function. So to guard against

incorrect usage of the contract functions a dedicated function modifier will check if a certain function can be called in a certain stage. The DEMO theory defines C-acts as acts in a business conversation, so these will be modeled as functions that can only be called by a certain address in a certain stage of the contract. To implement the guard of access to the functions the Restricting Access common pattern was implemented [32], through function modifiers once again. The P-act by the same logic is implemented through a function modifier that is only called in functions that represent the coordination act declares. The function modifier that represents the P-act will emit an event corresponding to the P-fact, as a production fact is a result of performing a production act. The model presented below refers to the contract from which all other contracts are derived.

```
contract Transaction {
    enum C_facts { Inital, Requested, Promissed, Declared, Accepted,
    Declined, Rejected }

    C_facts public c_fact = C_facts.Inital;

    address payable public initiator;
    address payable public  executor;

    event p_fact(address _from, bytes32 _hash);

    modifier p_act(){ bytes32 hash = keccak256(abi.encodePacked(now,
        block.difficulty, msg.sender));
        emit p_fact(msg.sender, hash);
        _; }

    modifier atCFact(C_facts _c_fact) {
        require(c_fact == _c_fact,
        "Function cannot be called at this time.");
        _; }

    modifier onlyBy(address _account) {
        require(msg.sender == _account, "Sender not authorized.");
        _; }

    function nextCFact(bool happyFlow) internal {
        if(happyFlow == true){
            c_fact = C_facts(uint(c_fact) + 1); }
        if(happyFlow == false && c_fact == C_facts.Requested){
            c_fact = C_facts.Declined; }
        if(happyFlow == false && c_fact == C_facts.Declared){
            c_fact = C_facts.Rejected; } }

    modifier transitionNext(bool happyFlow) {
            _;
            nextCFact(happyFlow); }
}
```

5 Use Case: Rent-A-Car

The case Rent-A-Car is an exercise in producing the essential model of an enterprise that offers the usufruct of tangible things: Rent-A-Car is a company that rents cars to customers. At [12] all four aspect models (CM, AM, PM, and FM) are presented. Together they constitute a coherent whole that offers full insight into an overview of the essence of car rental companies. The product action rules can directly be transformed into executable computer code.

After defining the parent contract, for each of the transaction kinds of the Rent-A-Car case, a contract is created. The idea would be for the client to deploy the *RentalCompleting* SC into the BC, after that he would be able to request that same transaction.

```
contract RentalCompleting is Transaction{
    struct Rental {
        uint256 stratingDate;
        uint256 endingDate;
        uint256 maxRentalDuration;
        uint256 drivingLicenseExpirationDay;
        ...}
    //other defined facts

    Rental public rental;
    //other declared  facts

    constructor() public{
        initiator = 0x5c80...; //rentACar
        executor = msg.sender; //client

        rental.maxRentalDuration = 10;
        //other initialized facts
    }
    ...
```

At the *requestRentalCompleting* the truth division of the assess part of ARS-1 is implemented through the build-in function *require()*. After all the required check and initializations the state of the *requestRentalCompleting* contract is changed to Requested.

```
function requestRentalCompleting
    (uint256 _startingDate, uint256 _endingDate,
    uint256 _drivingLicenseExpirationDay) public
        atCFact(C_facts.Inital) onlyBy(executor) {
            require(_endingDate >= _startingDate);
            //checks of truth division
            ...
        c_fact = C_facts.Requested; }
```

At the *promiseRentalCompleting* the contract DepositPaying is created. The DepositPaying contract must be deployed at the returned address. Note that in this particular case at the end of the function the state is not updated to

Promised as this will only be done at the *acceptInvoicePaying* function of the InvoicePaying contract as shown in the PSD at [10].

```
function promiseRentalCompleting() public
    atCFact(C_facts.Requested) onlyBy(initiator) returns (address) {
        DepositPaying depositPaying = new DepositPaying(address(this));
            return address(depositPaying); }
```

Only at the *requestDepositPaying* the With clause of the action clause of the response part of ARS-1 is implemented through the build-in *require()* function.

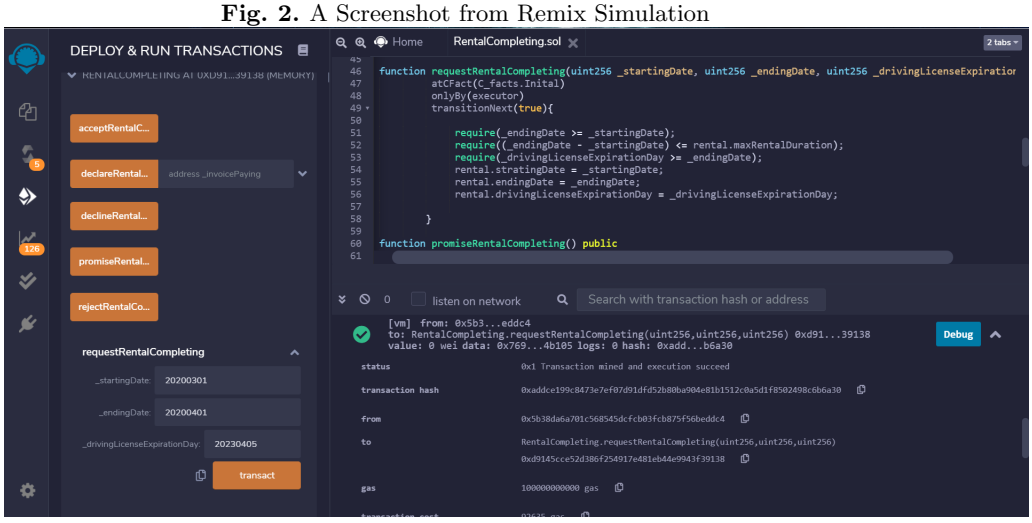
```
contract DepositPaying is Transaction {
    RentalCompleting rentalCompleting;

    //constructor

    function requestDepositPaying
    (uint256 _rq_depositAmount) public
    atCFact(C_facts.Inital) onlyBy(executor) {
        RentalCompleting.CarGroup memory cG
        = rentalCompleting.rental();
        require(_rq_depositAmount == cG.standardDepositAmount);
        c_fact = C_facts.Requested; }
```

Due to space restrictions, the rest of the code won't be shared but the logic throughout the transactions is always the same. It's also relevant to say that both the *declareDepositAmount* and *declareInvoicePaying* implement the Withdrawal common pattern. The rest of the generated code can be found on GitHub under `https://github.com/martasaparicio/try`.

Fig. 2. A Screenshot from Remix Simulation



6 Ontological Implications

With this fusion between DEMO Action Models and Smart Contracts, we believe multiple ontological changes would happen. These changes can be the disappearance of transactions that are tacitly performed by the Blockchain, making it difficult to distinguish between an executor role and an initiator role, or even the disappearance of some type of actor roles.

This Use Case presented may ease access to car renting to the different parties involved, especially the information exchange. Furthermore, this Use Case can benefit from the use of BC to provide an immutable trace of registry changes. The use of BC may also provide higher resilience to system faults and a more seamless exchange of funds.

Going even further, some ontological transactions could disappear, at least the ones that only exist to deal with truss issues between parties.

In this particular case, the *CarTaking* and *CarReturning* would be done tacitly, as the access to the asset, in this case a car, would be controlled by the BC. The control could be done through a publish subscribe mechanism. When the client invokes the functions *declareCarTaking()* and *declareCarReturning()*, that consequently invoke the modifier *p_fact()*, that in turn issues an event representing the *p_act()*. The Rent-A-Car may listen to these events by subscribing to them.

More generically, the execution of the functions representing the *declare* transaction step emit events, which are stored in the transactions tier of the blocks the transactions are contained in. All the other transaction steps are also recorded in the BC but without the emission of a representative event.

It is important to note, however, that in the solution presented human actors are ultimately responsible and accountable for the acts of these artifacts. Human actors send transactions to a BC via specific wallet software. The transaction contains a data payload containing instructions to invoke SC functions with specific input parameters. The function may update state variables, which are stored, in the contract's internal storage. Each *declare* function call may emit an event, to which an actor can subscribe. All other DEMO transaction steps, if successful, are also recorded in the BC.

Following the nomenclature suggested in [12], this solution does not remove the final responsibility and accountability for the acts from subjects. By subjects, it is understood, human beings responsible for actor roles. The technological capabilities of the BC, however, outperform, in order of magnitude, the subject's capabilities. This work proves the feasibility of implementing actor roles using BC. So, the question is, what it means to have agents perform the P-acts and C-facts while keeping in mind that agents cannot be held accountable. By agents understand BC artifacts to perform functions of, an actor role.

Transactions typically occur on one or more of the *performa*, *informa*, and *forma* levels. *Datalogical* transactions, given this work, do not present any contradiction in being executed by agents. However, there must always exist an actor ultimately responsible for the work and who can be held accountable for the agent. The solution presented makes the execution of P-acts tacit when

declaring a transaction. Since C-facts represent social commitments, C-acts are performed by the actor, himself through the invocation of a SC function. *Infological* transactions, in the perspective of this work, follow the same guideline as *Datalogical* transactions. The guideline proposed for *Datalogical* transactions allows exchanges of information or knowledge between actors. When it comes to performing original P-acts, at an *Ontological* transaction level, agents are not capable of dealing autonomously with it, as they cannot be ultimately responsible and accountable for the acts. However, BC can support O-actors with the P-acts by doing so as tacitly as possible, that is, by doing so by invoking the function that represents the declaration C-act. However, this form of operation must be known to the actor. This presented solution allows supporting O-actors, to a large extent, while never taking over the authority and responsibility that is assigned to an actor. This work defends the idea of co-existence between the subjects and the BC.

7 Conclusion and Further Research

The research presented in this paper is both timely and relevant as [4] research confirms an apparent synergy between artifact-centric process modeling and SCs.

To address this problem we propose a mapping between Solidity Concepts and DEMO Action Model concepts, this proposition can be seen as a way to implement DEMO Action Model. However, this work didn't consider optimization as an issue to resolve, for that reason, this must be considered as future work. Although this work presents a verification of the research presented through the Rent-A-Car Use Case a future work to consider would be related to its validation. Besides testing the mapping proposed in a more sizable sample of Action Models would be of great importance.

Modeling methods based on Ontology can lead to better data standards, business practices, and processes for developing and operating a Blockchain. The modeling methods based on Ontology can aid in the formal specifications for automated inference and verification in the operation of a Blockchain. This last benefit is particularly interesting as is very similar to the definition of Smart Contracts as "pieces of software that represent a business arrangement and execute themselves automatically under predetermined circumstances" [33]. For these reasons, Kim and Laskowski claimed in [34] that "ontology-based blockchain modeling will result in a blockchain with enhanced interpretability" [34].

References

1. M. Op't Land, E. Proper, M. Waage, J. Cloo, and C. Steghuis,Enterprise architecture: creating value by informed governance. Springer Science & Business Media, 2008.
2. M. Dumas, M. L. Rosa, J. Mendling, and H. A. Reijers,Fundamentals of Business Process Management. Springer Berlin, 2019.

3. K. C. Laudon and C. G. Traver, E-commerce: business. technology. society. Pearson, 2020.
4. M. van Wingerde and H. Weigand, "An ontological analysis of artifact-centric business processes managed by smart contracts," in 2020 IEEE 22nd Conference on Business Informatics (CBI), vol. 1.IEEE, 2020, pp. 231–240.
5. M. Muller, N. Ostern, and M. Rosemann, "Silver bullet for all trust issues? blockchain-based trust patterns for collaborative business processes,"Blockchain for Trust-aware Business Processes Preprint, 07 2020.
6. Gartner,"The reality of blockchain,"https://www.gartner.com/smarterwithgartner/the-reality-of-blockchain/, 2019, accessed: 2020-08-10.
7. Gartner,"Gartner survey reveals the scarcity of current blockchain deployments,"https://www.gartner.com/en/newsroom/press-releases/2018-05-03-gartner-survey-reveals-the-scarcity-of-current-blockchain, 2018, accessed: 2020-08-10.
8. A. Norta, "Designing a smart-contract application layer for transacting decentralized autonomous organizations," in Advances in Computing and Data Sciences, M. Singh, P. Gupta, V. Tyagi,A. Sharma, T. Oren, and W. Grosky, Eds. Singapore: Springer Singapore, 2017, pp. 595–604.
9. X. Xu, I. Weber, and M. Staples, "Model-driven engineering for blockchain applications," in Architecture for Blockchain Applications. Springer, 2019, pp. 149–172.
10. J. L. G. Dietz, Enterprise ontology: theory and methodology. Springer, 2011.
11. M. Andrade, D. Aveiro, and D. Pinto, "Demo based dynamic information system modeller and exe-cuter,"Proceedings of the 10th International Joint Conference on Knowledge Discovery, KnowledgeEngineering and Knowledge Management, 2018.
12. J.L.G. DIETZ, J.B.F. MULDER,ENTERPRISE ONTOLOGY: a human-centric approach to understanding the essence of organisation. SPRINGER NATURE, 2020.
13. J. Dietz and J. Hoogervorst, "Enterprise ontology and enterprise architecture–how to let them evolve into effective complementary notions,"GEAO Journal of Enterprise Architecture, vol. 2, no. 1, pp.121–149, 2007.
14. Hevner, A. (2007). A Three Cycle View of Design ScienceResearch.Scandinavian Journal of Information Sys-tems, 19(2). http://aisel.aisnet.org/sjis/vol19/iss2/4.
15. Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). De-sign Science in Information Systems Research.MISQ., 28(1), 75–105.
16. Aparicio., M., Guerreiro., S., & Sousa., P. (2020). Towards an automated demo action model implementation using blockchain smart contracts. InProceedings of the 22nd International Conference on Enterprise Information Systems - Volume 2: ICEIS,(pp. 762–769).:INSTICC SciTePress.
17. Frantz, C. K. and Nowostawski, M. (2016).From institutions to code: Towards automated generation ofsmart contracts. In2016 IEEE 1st International Work-shops on Foundations and Applications of Self* Systems (FAS*W), pages 210–215.
18. Crawford, S. E. S. and Ostrom, E. (1995). A grammar of institutions.American Political Science Review,89(3):582–600.
19. Mavridou, A. and Laszka, A. (2018).Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach, pages 523–540.
20. Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev,A., and Mendling, J. (2016). Untrusted business process monitoring and execution using blockchain. In La Rosa, M., Loos, P., and Pastor, O., editors,Busi-ness Process Management, pages 329–347, Cham.Springer International Publishing.
21. Garcia-Banuelos, L., Ponomarev, A., Dumas, M., and Weber, I. (2017). Optimized execution of business processes on blockchain.

22. Pintado, O. (2017). Caterpillar: A blockchain-based business process management system.
23. Pintado, O., Garciaa-Banuelos, L., Dumas, M., Weber, I.,and Ponomarev, A. (2018). Caterpillar: A business process execution engine on the ethereum blockchain.
24. Tran, A. B., Lu, Q., and Weber, I. (2018). Lorikeet: A model-driven engineering tool for blockchain-based business process execution and asset management. In BPM.
25. Nakamoto, S. (2009).Bitcoin: A peer-to-peer elec-tronic cash system.Cryptography Mailing list athttps://metzdowd.com.
26. Bahga, A. and Madisetti, V. (2016). Blockchain platformfor industrial internet of things.Journal of SoftwareEngineering and Applications, 09:533–546.
27. D. C. Schmidt, "Guest editor's introduction: Model-driven engineering,"Computer, vol. 39, no. 2, pp.25–31, Feb 2006.
28. B. Scott, J. Loonam, and V. Kumar, "Exploring the rise of blockchain technology: Towards distributed collaborative organizations,"Strategic Change, vol. 26, no. 5, pp. 423–428, 2017.
29. "Why model-driven engineering fits the needs for blockchain application development." [Online].
30. Guerreiro, S. (2012). Enterprise dynamic systems control enforcement of run-time business transactions using demo: principles of design and implementation. Instituto Superior Tecnico, Lisboa.
31. Skotnica, M., van Kervel, S. J. H., and Pergl, R. (2017).A demo machine - a formal foundation for executionof demo models. In Aveiro, D., Pergl, R., Guizzardi,G., Almeida, J. P., Magalhaes, R., and Lekkerkerk, H.,editors,Advances in Enterprise Engineering XI, pages18–32, Cham. Springer International Publishing.
32. Wohrer, M. and Zdun, U. (2018).Design patterns forsmart contracts in the ethereum ecosystem. In2018IEEE International Conference on Internet of Things(iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data(SmartData), pages 1513–1520.
33. "Not-so-clever contracts." [Online].
34. H. Kim and M. Laskowski, "Towards an ontology-driven blockchain design for supply chain prove-nance," inWiley Online Library, 08 2016.