Automated DEMO Action Model Implementation using Blockchain Smart Contracts

Marta Aparício[1,2], Sérgio Guerreiro[1,2] and Pedro Sousa[1,2,3]
[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*

Keywords: Blockchain, DEMO, DEMO Action Model, Ethereum, Smart Contract.

Abstract: Enterprise Ontology theory describes a well-founded method to model the essence of an organization in a coherent, comprehensive, consistent, and concise way. Enterprise Ontology can offer advantages in understanding the essence of an organization and in using organization models as a starting point for building software supporting organizations. The availability of ontological models that express the essence of an organization becomes the fundamental element to support the correct implementation of Smart Contracts in the Blockchain of that same organization. In this context, it is intended to automatically extract from the DEMO Action Model the knowledge necessary to produce Smart Contracts in Blockchain. The advantage to be obtained is the reuse of the modeling done ontologically in line with a correct implementation of the Smart Contracts. This research feasibility is demonstrated through the well-known Rent-A-Car case.

1 INTRODUCTION

According to (Dietz and Hoogervorst, 2007) two different system notions exist, the teleological and the ontological system notion. The teleological system notion is about the function and the behavior of a system. This notion can be visualized with a black-box model. The ontological system notion, on the other side, is about the construction and operation of a system and can be modeled with a white-box model. Both the teleological and the ontological system notions are relevant for designing a system.

The starting point in designing a system is the using system (US). From the construction (white-box model) of the US it can be determined the requirements for the object system (OS). These requirements are, by nature, about the function and behavior of the OS, thus in terms of the black-box model of the OS (Dietz and Hoogervorst, 2007). If the design processed by the system design process is specified as an ontology, the process of engineering a system is like the one shown in Figure 1. An ontology model of a system is fully independent of the implementation, it only shows the essential features. A good example of such ontology is the DEMO (Design & Engineering Methodology for Organizations) approach to enterprise ontology, considering enterprises also as

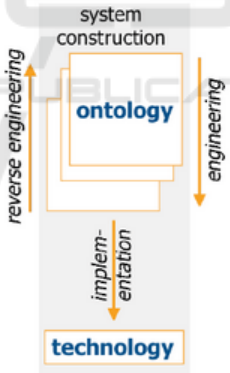

Figure 1: The process of engineering a system (Dietz and Hoogervorst, 2007).

systems (Dietz, 2006). The implementation model, also known as Action Model, can straightforwardly be implemented on the available technological platform (Dietz and Hoogervorst, 2007). In software development this model often is the source code in a programming language like Solidity. Smart Contract (SC) is an application of the Blockchain (BC) technology to create an independently verifiable, secure, permanent, and fault-tolerant agreement designed for satisfying common contractual conditions. Enterprise organizations need to establish a level of trust among their users purchasing products or services. That can

Aparício, M., Guerreiro, S. and Sousa, P.
Automated DEMO Action Model Implementation using Blockchain Smart Contracts.
DOI: 10.5220/0010147602830290
In *Proceedings of the 12th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2020) - Volume 2: KEOD*, pages 283-290
ISBN: 978-989-758-474-9

be accomplished via a SC agreement. The availability of ontological models that express the essence of an organization becomes the fundamental element to support the correct implementation of SCs in the BC of that same organization. In this context, it is intended to automatically extract from the DEMO Action Model the knowledge necessary to produce SCs in BC. The advantage to be obtained is the reuse of the modeling done ontologically in line with a correct implementation of the SCs.

There are some works that relate ontology and BC, such as Understanding the BC Using Enterprise Ontology (de Kruijff and Weigand, 2017), and Exploring a Role of BC SCs in Enterprise Engineering (Hornáčková et al., 2019). The intersection between SCs and Ontology, is seen as a great opportunity since an ontology-based BC will have an enhanced interpretability, when compared with a more traditional way of development and management (Kim and Laskowski, 2016). However, there is no clear mapping between Enterprise Ontology and BC. These statements lead to the following research questions: *What is the correspondence between Blockchain Smart Contracts and DEMO Action Model?* and *How can we generate DEMO Action Model from Blockchain Smart Contracts?*

2 BACKGROUND

2.1 Demo Theory

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 (Andrade et al., 2018). 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 are guaranteed, even though they're abstract and have a conceptual nature. The core concept of DEMO is a transaction, fully based on the $\psi - theory$.

According to $\psi - theory$, in the standard pattern of a business transaction exists two actor roles, the initiator and the executor. The obtained fact when performing a business transaction, is originated by the collaboration of production and coordination acts. These acts contain three phases each one with specific steps. (1) The Order phase that contains the request (rq); promise (pm); decline (dc); quit (qt) steps. (2) The Execution phase that contains only the execution (ex) step. (3) The Result phase that contains the states (st); reject (rj); accept (ac) steps. The following four steps are present in every transaction and represent the happy flow: request, promise, state, accept.

2.2 Blockchain

(Nakamoto, 2009) described BC as an architecture that gives participants the ability to perform electronic transactions without relying on trust. What makes this possible it 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 an 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 as the capacity of calculate hundreds of thousands of hashes, per second (Aparício. et al., 2020). 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.

2.3 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 automat-

ically without discretion. The trust issue is also addressed, once Smart Contrats 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 isn't possible to tamper 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 cannot force the behavior of the contract since it is dependent of the other nodes. Like all algorithms SCs may require input values, and only act if certain predefined 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 inputed to the system, then false results will be outputed (Bahga and Madisetti, 2016).

3 RELATED WORK

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. The Agent-Based Approach described by (Frantz and Nowostawski, 2016) proposes a modeling approach that supports the semi-automated translation of human-readable contract representations into computational equivalents in order 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 in order to support the contract modeling process. The concept Grammar of Institutions (Crawford and Ostrom, 1995) 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 **A**ttributes describing an actor's characteristics or attributes. The **D**eontic describing the nature of the statement as an obligation, permission or prohibition. The A**I**m describing the action or outcome that this statement regulates. The **C**onditions describing the contextual conditions under which this statement holds. And the **O**r else describing consequences associated with non-conformance. Using these components, statements on the execution of the SC are made. The statements are then linked by the structure of Nested ADICO (Boella et al., 2013), a variant of ADICO in which the institutional functions are linked by the operators AND, OR, and XOR to create a simple set of prescriptions. The set of prescriptions is then transformed into a contract skeleton which has to be finished manually. Furthermore, it is argued that the Grammar of Institutions invites non-technical people to the SC development process.

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. (Mavridou and Laszka, 2018) show that the transformation of the Finit State Machine to Solidity is partly automated, since to ensure Solidity code quality, some manual coding might be necessary or added though plugins.

For Process-Based Approaches, both DEMO and Business Process Model and Notation (BPMN) are well established for modelling business processes. (Weber et al., 2016), describes a proposal to support inter-organizational processes through BC technology. Captured in BPMN, large parts of the control flow and business logic of inter-organizational business processes can be compiled from process models into SCs that ensure that the joint process is correctly executed. So-called trigger components allow the connection of these inter-organizational process implementations to Web services and internal process implementations. These triggers serve as a bridge between the BC and enterprise applications. Basically, Weber et al. developed a technique to integrate BC into the choreography of processes in order to maintain trust. The BC enabled to store the status of process execution across all involved participants, as well as to coordinate the collaborative business process execution. Validation was made against the ability to distinguish between conforming and non-conforming traces. (García-Bañuelos et al., 2017) presents an optimization for (Weber et al., 2016). In this work, to compile BPMN models into a SC 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 SC. Compared to (Weber et al., 2016) this work (García-Bañuelos et al., 2017) managed to decrease the amount paid of resources and achive a higher throughput. Caterpillar, first presented in (Pintado, 2017) and further discussed in (Pintado et al., 2018), is an open-source Business Process Management System (BPMS) 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 SCs generated by a BPMN-to-Solidity compiler. Caterpillar implements a comprehensive mapping from BPMN to Solidity. Given a BPMN model (in standard XML format), it generates a SC (in Solidity), which encapsulates the workflow routing logic of the process model. Specifically, the SC contains variables to encode the state of a process instance, and scripts to update this state whenever a task completes or an event occurs. Caterpillar supports not only basic BPMN control flow elements (i.e. tasks and gateways), but also includes advanced ones, such as sub-processes, multi-instances and event handling. Lorikeet (Tran et al., 2018), on other hand can automatically create well-tested SC 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 SCs in Solidity from BPMN models while the registry generator creates Solidity SC based on the registry models. The BPMN translator takes an existing BPMN business process model as input and outputs a SC. 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 SC. Users can then deploy the SCs on BC. This work builds up on already seen works, such as (García-Bañuelos et al., 2017) (Weber et al., 2016), for the BPMN translation algorithms.

4 PROPOSED SOLUTION

As explained by Alex Norta in (Norta, 2017), referencing a crowdfunding project that was hacked as it contained security flaws, resulting in a considerable monetary loss. *"The incident shows it is not enough to merely equip the protocol layer on top of a Blockchain 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"*. A solution to this referred problem would be to generate SCs automatically, which would add a level of security.

4.1 Solution Hypotheses

The concept of data independence designates the techniques that allow data to be changed without affecting the applications that process it (Markov, 2008). It is the ability to modify a scheme definition in one level without affecting a scheme definition in a higher level. It is believe that a similar separation is highly needed for SCs, in order to achieve the goals set in this work. DEMO is based on explicit specified axioms characterized by a rigid modeling methodology, and is focused on the construction and operation of a system rather than the functional behavior. It emphasizes the importance of choosing the most effective level of abstraction during information system development, in order to establish a clear separation of concerns. The adoption of the *Distinction Axiom* of Enterprise Ontology, presented in section 2.1, is proposed as an ontological basis for this separation.

In a first approach to the problem, it is believed that, for the Datalogical Layer a SC can be defined as a piece of code contained on a node of the BC. For the Infological layer a SC is enforced by a set of rules implemented on the BC through code. And lastly, for the Essential layer, the SC is a contract in which the commitment fulfillment is completely or partially performed automatically in BC. This step allows to defend Ontology, and in particular DEMO, as a good way towards a model-driven approach in regards to the automatic generation of software artifacts.

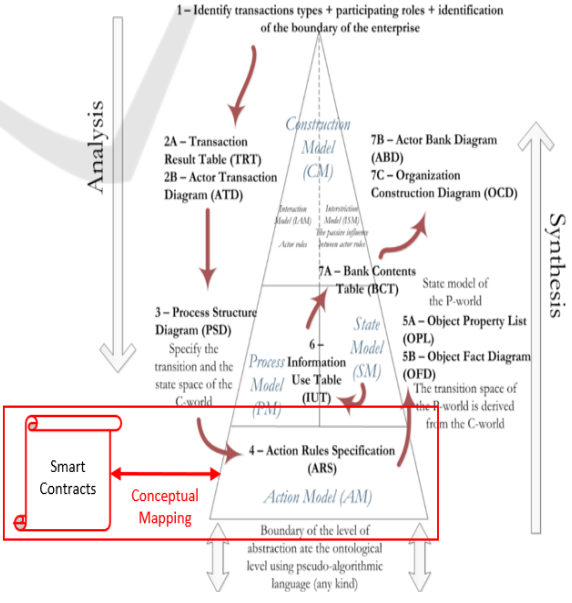

Figure 2: Proposed Solution Architecture, original figure from (Guerreiro, 2012).

4.2 Solution Architecture

This work start with a believe that a DEMO transaction is represented as a contract in a BC. The contract has its own address, internal storage, attributes, methods and it is callable by either an external actor or another contract, as mentioned in section 2.3. This is the functionality needed to represent a DEMO transaction. They implemented the execution of DEMO transactions according to the DEMO Machine (Skotnica et al., 2017) and associated theories. Now, this present work argues that the SCs automated generation could be done directly form the Action Model with no need for the generation of the remaining DEMO models. 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 models, the basis of the DEMO methodology is exactly the *Action Model*, as can be seen in Figure 2. 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 transaction 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 2 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 in a different way. 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.

4.3 Generate DEMO Action Model from Blockchain Smart Contracts

In this section, the implementation of the Action Rules of the Action Model is proposed by means of SCs. This is enabled by the ability of SCs within the scope of BC technology to describe complex algorithms by using the Turing-complete programming language.

Solidity is a high-level programming language to implement SCs specially design for the Ethereum Virtual Machine (EVM). Solidity was chosen because it is developed under Ethereum and it is the most used language for SCs for EVM. The building block in Solidity is a contract which is similar to class in object-oriented programming. Contract contains persistent data in state variables, functions to operate on this data and it also supports inheritance. Contract can further contain function modifiers, events, struct types and other structures to allow 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 create. Either way the contract code is than compiled to the EVM bytecode, new transaction is created holding the code and deployed to BC, returning the address of the contract for further interaction.

The proposed mapping not only takes advantage of the intrinsic properties of BC technology but also takes advantage of some design patterns for SCs in the Ethereum Ecosystem (Wöhrer and Zdun, 2018).

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. 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, this stages are implemened as Enums. Enums are a way to create a user-defined type in Solidity, for this particular application seven stages, coresponding to the coordination facts: Inital; Requested; Promissed; 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 automati-

cally check a condition prior to 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, 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 declare. The function modifier that represents the P-act will emit an event corresponding to the P-fact, as a production fact is the result of performing a production act. To summarize, Table 1 shows the proposed correspondence.

Table 1: Correspondence between Solidity SCs and DEMO AM concepts.

DEMO component	Solidity
C-act	Enum
C-fact	Funtion
P-act	Funtion Modifier
C-fact	Event

5 CASE STUDY: RENT-A-CAR

In this section, the implementation of the Action Model of a well-known case in the enterprise ontology is attempted. 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 (Dietz and Mulder, 2020) all four aspect models (CM, AM, PM, and FM) are presented. Together they constitute a coherent whole that offers full insight into and overview over the essence of car rental companies. The produced action rules can directly be transformed into executable computer code.

As a starting point a generic Transaction contract was created, from which all other transactions derive.

```
contract Transaction {
    enum C_facts {
        Inital,
        Requested,
        Promissed,
        Declined,
        Declared,
        Accepted,
        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));
        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.");
        _;} }
```

As discussed in Section 4.3 the c-facts are represented through an enum, with the seven stages considered as the coordination facts. The initiator and executioner are represented through each of their addresses. The *p-act()* is represented by a function modifier that emits the p-fact event. At last, the *atCFact()* and *onlyBy()*, are respectively modifiers of the common patterns State Machine and Restricting Access.

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 *re-*

questRentalCompleting 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 Promissed as this will only be done at the *acceptInvoicePaying* funtion of the InvoicePaying contract as showed in the PSD at (Dietz and Mulder, 2020).

```
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; }
```

After the promiseDepositAmount, the declareDepositAmount implements once again a common pattern in solidity, the Withdrawal from Contracts as this is the recommended method of sending funds. Only the declareDepositAmount and declareInvoicePaying implement this pattern. Also note that this must be a payable function and *p-act()* modifier must be present.

For a better understanding of the sequence systematization table 2 is presented.0

Table 2: Sequence systematization of Rental-A-Car case.

RentalCompleting	
rq	with clause of event part (ARS-1)
pm	
DepositPaying	
rq	with clause of response part (ARS-1)
pm	
da	
ac	truth division of assess part (ARS-2)
CarTaking	
rq	with clause of response part (ARS-3)
pm	
da	
ac	truth division of assess part (ARS-4)
...	
InvoicePaying	
rq	with clause of response part (ARS-7)
pm	
da	
ac	truth division of response part (ARS-8)
RentalCompleting	
da	
ac	

6 CONCLUSIONS

The research presented in this paper is both timely and relevant as (van Wingerde and Weigand, 2020) research confirms an apparent synergy between artifact-centric process modeling and SCs. SCs essentially consist of two elements, programmable logic and data storage. With these two elements, organizations may model business logic and data models of their shared processes. One of BC' main objectives is to handle this massive amount of complex contractual agreements and transactions that are nowadays created. Also, intends to free parties that are dependent upon third parties to manage and enforce those contractual agreements. However, non-technical people, that do not comprehend software code are once again dependent on a third party to write them contracts. The low level of semantics of software code makes it challenging to have a high level of comprehension and reasoning, which makes it more prone to errors. There are already some research towards raising the level of abstraction from code-centric to model-centric SCs development. To address this problem we propose a conceptual 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 considered optimization as an issue to resolve, for that reason this must be considered as a future work. Although this work presents a

verification of the research presented though the Rent-A-Car 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.

ACKNOWLEDGEMENTS

This work was supported by the European Commission program H2020 under the grant agreement 822404 (project QualiChain) and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020 (INESC-ID).

REFERENCES

Andrade, M., Aveiro, D., and Pinto, D. (2018). Demo based dynamic information system modeller and executer. pages 383–390.

Aparício., M., Guerreiro., S., and Sousa., P. (2020). Towards an automated demo action model implementation using blockchain smart contracts. In *Proceedings of the 22nd International Conference on Enterprise Information Systems - Volume 2: ICEIS,*, pages 762–769. INSTICC, SciTePress.

Bahga, A. and Madisetti, V. (2016). Blockchain platform for industrial internet of things. *Journal of Software Engineering and Applications*, 09:533–546.

Boella, G., Elkind, E., Savarimuthu, B. T. R., Dignum, F., and Purvis, M. K. (2013). Prima 2013: Principles and practice of multi-agent systems. In *Lecture Notes in Computer Science*.

Crawford, S. E. S. and Ostrom, E. (1995). A grammar of institutions. *American Political Science Review*, 89(3):582–600.

de Kruijff, J. and Weigand, H. (2017). Understanding the blockchain using enterprise ontology.

Dietz, J. (2006). *Enterprise Ontology—Theory and Methodology*.

Dietz, J. and Hoogervorst, J. (2007). Enterprise ontology and enterprise architecture–how to let them evolve into effective complementary notions. *GEAO Journal of Enterprise Architecture*, 2(1):121–149.

Dietz, J. L. and Mulder, H. B. (2020). *Enterprise Ontology: A Human-Centric Approach to Understanding the Essence of Organisation*. Springer Nature.

Frantz, C. K. and Nowostawski, M. (2016). From institutions to code: Towards automated generation of smart contracts. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 210–215.

García-Bañuelos, L., Ponomarev, A., Dumas, M., and Weber, I. (2017). Optimized execution of business processes on blockchain.

Guerreiro, S. (2012). Enterprise dynamic systems control enforcement of run-time business transactions using demo: principles of design and implementation. Instituto Superior Técnico-Universidade Técnica de Lisboa, Lisboa.

Hornáčková, B., Skotnica, M., and Pergl, R. (2019). Exploring a role of blockchain smart contracts in enterprise engineering. In Aveiro, D., Guizzardi, G., Guerreiro, S., and Guédria, W., editors, *Advances in Enterprise Engineering XII*, pages 113–127, Cham. Springer International Publishing.

Kim, H. and Laskowski, M. (2016). Towards an ontology-driven blockchain design for supply chain provenance.

Markov, K. (2008). Data independence in the multi-dimensional numbered information spaces.

Mavridou, A. and Laszka, A. (2018). *Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach*, pages 523–540.

Nakamoto, S. (2009). Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*.

Norta, A. (2017). Designing a smart-contract application layer for transacting decentralized autonomous organizations. In Singh, M., Gupta, P., Tyagi, V., Sharma, A., Ören, T., and Grosky, W., editors, *Advances in Computing and Data Sciences*, pages 595–604, Singapore. Springer Singapore.

Pintado, O. (2017). Caterpillar: A blockchain-based business process management system.

Pintado, O., García-Bañuelos, L., Dumas, M., Weber, I., and Ponomarev, A. (2018). Caterpillar: A business process execution engine on the ethereum blockchain.

Skotnica, M., van Kervel, S. J. H., and Pergl, R. (2017). A demo machine - a formal foundation for execution of demo models. In Aveiro, D., Pergl, R., Guizzardi, G., Almeida, J. P., Magalhães, R., and Lekkerkerk, H., editors, *Advances in Enterprise Engineering XI*, pages 18–32, Cham. Springer International Publishing.

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*.

van Wingerde, M. E. M. and Weigand, H. (2020). An ontological analysis of artifact-centric business processes managed by smart contracts. In *2020 IEEE 22nd Conference on Business Informatics (CBI)*, volume 1, pages 231–240.

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, *Business Process Management*, pages 329–347, Cham. Springer International Publishing.

Wöhrer, M. and Zdun, U. (2018). Design patterns for smart contracts in the ethereum ecosystem. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1513–1520.