Implementation of DEMO Action Model in Blockchain Smart Contracts

Marta Maria Simões Aprício
Instituto Superior Técnico, University of Lisbon, Av. Rovisco Pais 1, 1049-001 Lisbon, Portugal
marta.m.s.aparicio@tecnico.ulisboa.pt

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

Abstract: The intricate environment created by global trading leads to enforcing trust by centralized organizations. A potential problem is that these organizations constitute a single point of trust and failure. Blockchain (BC) offers a decentralized ledger that records transactions and allows tracking assets in a secure and reliable way. While BC can guarantee that the stored data cannot be tampered with, it cannot guarantee that the data was correct when it was stored in the chain. A correctness check for data before data is stored in the BC is needed. The DEMO methodology offers all the information about the process needed to evaluate and create Smart Contracts (SC). By computing a high-level formal model as DEMO Action Model and generating SCs from it, one can create SCs that ensure the correctness of transactions data before it is stored in the BC. This solution allows the reuse of Ontological models and guarantees the correct implementation of SCs. This work relates concepts between DEMO and Solidity and creates a base SC that encapsulates the DEMO concepts. Regarding the mapping between the DEMO concepts and Solidity concepts, DEMO ensures the syntactic validation, while the two Use Case ensure the semantic validation. The base contract, it instantiates the mapping, and its validation is done using the EVM compiler and Solidity language. The work was communicated to ICEIS, KEOD, and MODELSWARD to obtain a scientific evaluation.

1 INTRODUCTION

BC technology can be seen as a solution to coordinate inter-organizational processes involving untrusted parties (Dumas et al., 2019). This technology provides a way to record something that happened to ensure that it cannot be deleted once recorded. SCs 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.

While BC can guarantee that the stored data cannot be tampered with, it cannot guarantee that the data was correct when it was stored in the chain. Rectifying incorrect information is almost impossible in the BC. What is needed is a correctness check for data before data is stored in the BC.

Dietz uses the ψ-theory (Dietz, 2007) (underlies the notion of Enterprise Ontology (EO)) 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 (Dietz, 2006). This methodology is called DEMO. 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.

The EO model reduces complexity. This reduction in complexity makes the organization moretransparent. It also shows the consistency between all areas within the enterprise, such as Business Processes (BP) and workflows.

Concerning the automatic generation of SCs, 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.

By computing a high-level formal model as DEMO Action Model and generating SCs from it, one can create SCs that ensure the correctness of transactions data before it is stored in the BC. This solution allows enterprises to experience the possibilities of BC without needing to know all sorts of technical details. Ultimately, this solution allows the reuse of Ontological models and guarantees the correct imple-

mentation of SCs.

2 PROPOSAL

In this section the artifacts developed in the scope of this work are presented.

2.1 Ontological Solution Hypothesis

The concept of data independence designates the techniques that allow data to change without affecting the applications that process it (Markov, 2008). It is the ability to modify a scheme definition at one level without affecting a scheme definition at a higher level. It is believed that a similar separation is highly needed for SCs to achieve the goals set in this work. DEMO is based on explicitly 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 to establish a clear separation of concerns. The adoption of the *Distinction Axiom* of EO is proposed as an ontological basis for this separation.

At the *Essential* layer, an inter-organizational BP consists of several causally related transactions, as depicted by the *Composition Axiom* of EO. Each of these transactions has one initiator and one executor, which are roles played by process participants. Process participants perform acts during the various transaction phases. Whenever an act is carried out, it results in a fact. Whereas an act typically consists of a desirable future state, a fact states something true in the social world at a point in time (van Wingerde and Weigand, 2020). In this layer, the present work believes a SC is a contract in which the commitment fulfillment is completely or partially performed automatically in BC.

The acts on the *Essential* layer maps to one or more invocations to a contract on the *Infological* layer. This contract guards the invocation of business rules that can potentially change the state of that same contract. So, this work believes, for the *Infological* layer, a SC is enforced by a set of rules implemented on the BC through code.

On the *Datalogical* layer, an inter-organizational process is invocations to SCs made by wallets. A transaction invocation, if successful, is recorded in a block. The execution of the function may emit an event and change state variables. So, for the *Datalogical* Layer, a SC is defined as a piece of code contained in nodes of the BC.

2.2 Modeling Solution Hypothesis

One of the most common graphic process modeling notations used is BPMN. BPMN focuses on modeling the articulation between activities, resources, flows, gateways, events, messages, and data objects that occur in a BP. However, BPMN doesn't adequately support the articulation of business rules that would be essential for the automatic generation of SCs. Process modelers usually have to use workarounds, usually by also using additional tools, to include business rules in their processes. In addition, (Tobias and Jan, 2013), concluded that BPMN has a level of 51.3% of overlapping language concepts and lacks state concept to ensure a more sound semantics.

DEMO, on the other hand, has been widely accepted in both scientific research and practical appliance (Andrade et al., 2018; DIETZ, 2020). DEMO, sees an enterprise as a system of people and their relations, authority, and responsibility. DEMO ψ-theory specifies semantic meaning to the business transactions. The business transactions dynamic includes the actors, the communications, the productions, and all its dependencies. While BPMN does not describe any semantic for the business model, it only provides a set of constructs that could be combined accordingly with a specification.

2.3 Technology used in Solution Implementation

This work requires that the BC chosen supports the creation of SCs.

2.3.1 Ethereum

Ethereum is a platform for BC applications, with its BC and cryptocurrency, *Ether*.

Ethereum does this by building what is essentially the ultimate abstract foundational layer: a BC with a built-in Turing-complete programming language, allowing anyone to write SCs where they can create their own arbitrary rules for ownership, transaction formats, and state transition functions.

2.3.2 Solidity

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 that is similar to a class in object-oriented programming. The contract contains persistent data in state variables, functions to operate on this data, and it also supports inheritance. The 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.

2.3.3 Remix

Remix is a browser-based IDE for creating SCs with an integrated debugging and testing environment. Remix offers development, compilation, and deployment of solidity contracts as well as access to already deployed contracts. The testing environment allows running the transactions in a sandbox BC in the browser with JavaScript VM with a possibility to switch between virtual accounts and spend virtual *Ether* for full SC testing.

2.3.4 MetaMask

Metamask is what makes it possible to switch between virtual accounts and spend virtual *Ether* for full SC testing. MetaMask is a browser plugin that serves as an Ethereum wallet. It allows users to store *Ether*, enabling them to make transactions to any Ethereum address.

2.4 Solution Architecture

This work starts with the belief a DEMO transaction is represented as a contract in a BC. The contract has its 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. Now, this present work argues that the DEMO *Action Model* could be implemented through BC SCs without any support of the remaining DEMO models. It believes that the structure and content of a SC can be directly mapped to each action rule. By creating the action rules, the logic in which the SC operates is also being created.

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 1. The *Cooperation 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

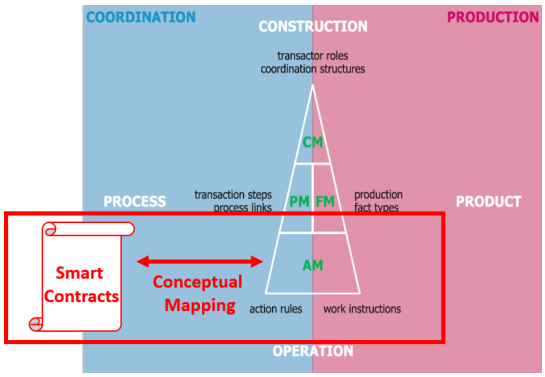

Figure 1: Solution Architecture.

of the triangle it is suggested that is the most concise model. The *Process Model* contains, for every transaction type in the *Cooperation 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 *Cooperation Model* in the triangle because it is the first level of detailing of the *Cooperation 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 *Cooperation Model*, namely, the detailing of the identified steps in the *Process Model* of the transaction types in the *Cooperation 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 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 *Cooperation Model*, *Process Model*, and *Fact Model*; but in a different way. These models have as if a zoom in (*Action Model*) zoom out (*Cooperation Model*) relationship between each other. The *Action Model* is the most detailed and comprehensive aspect model.

2.5 Generate DEMO Action Model from Blockchain Smart Contracts

The implementation of the Action Rules of the *Action Model* is proposed using 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.

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 invoked. A function call often ends a stage and transitions the contract into the next stage, this is known as the State Machine common pattern (Wöhrer and Zdun, 2018). Through the DEMO theory, it is known that actors interact through creating and dealing with coordination facts (C-fact). 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 C-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 coordination acts (C-act) 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 (Wöhrer and Zdun, 2018), through function modifiers once again. The production act (P-act) by the same logic is implemented through a function modifier that is only called in functions that represent the C-act *declare*. The function modifier that represents the P-act will emit an event corresponding to the production fact (P-fact), as a production fact is a result of performing a P-act.

To summarize, 1 shows the proposed correspondence.

Table 1: Correspondence between Solidity Smart Contract Components and DEMO Components.

DEMO Components	Solidity Components
C-fact	Enum
C-act	Function Invocation
P-fact	Event
P-act	Function Modifier

```
pragma solidity >=0.4.22 <0.7.0;
contract Transaction {
    enum C_facts {
    Initial, Requested, Promissed, Declared,
    Accepted, Declined, Rejected }
    C_facts public c_fact = C_facts.Initial;
    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 now.");
        _;                                }
    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);           }
}
```

Regarding the mapping presented in Table 1, the syntactic validation is guaranteed by DEMO, while the semantic validation is done by two Use Cases presented.

Since the building block in Solidity is a contract, that compares to a class in Object-Oriented programming. In a more Object-Oriented manner, the Transaction Contract can be interpreted as the Base-Class from which the Sub-Classes inherit some of its functionalities.

The abstraction created allows for the contracts, seen as Sub-Classes, to re-use code from the Base-Class (Transaction Contract). Besides, the re-use of the functionalities allows for the Sub-Classes to only implement the business logic associated with it.

As discussed, the *C-facts* are represented through an Enum, with the seven stages considered as the coordination facts. The *initiator* and *executor* 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, already presented before. The *transitionNext()* modifier uses the *nextC-*

Fact() function to control to switch between *C-facts*, checking if the execution followed an happy flow or not. The _ symbol present inside the modifiers in Transaction Contract is a place holder. The function body is inserted where the special symbol _ in the definition of a modifier appears. This functionality allows executing modifiers after the correct execution of a function, as the *transitionNext()*, or before, as *onlyBy()*. By defining the *initiator* and *executor* as *address payable*, at run-time this type of EOA allows for exchange of *Ether*, having access to primitives useful to manage *Ether*.

The deployment of the contract presented in Transaction Contract, takes 6 seconds and costs 0.000646 *Ether* (0,43 $). This contract only offers access to the following variables: *C_facts*, *initiator*, and *executor*. This contract doesn't give access to any function. Meaning, this contract encapsulates the DEMO concepts, but by itself, it is not very useful as it has no business logic.

SCs can store data and attest transaction execution. From the *Action Model*, it's possible to retrieve which object classes the transaction needs and at which point they arise at the transaction execution, besides evaluating the changes of the objects associated with the transaction execution. Contract internal state variables represent object classes, with the corresponding DEMO name. The SC then serves as a database for the transaction.

To attest transaction execution contracts represent all possible C-facts. The contract's then hold their current status as C-facts. For every C-act, a contract method exists that changes the contract state to the corresponding C-fact. Every change of C-fact issues an Ethereum system-wide notification, allowing external systems to keep track of their contracts.

The action rules of the *Action Model* define the operations for each actor role. The contract execution logic results from the translation of action rule pseudo-code. The name of the functions correspondent of each C-act is the C-act concatenated with the transaction name.

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 (DIETZ, 2020), 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 dissertation 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, in view of 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 obviously 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 dissertation defends the idea of co-existence between the subjects and the BC.

The sub-transactions are implemented as other contracts. In DEMO, there are two kinds of dependencies between BP: *(i)* Request after Promise, request of a new BP is triggered after a promise of a previous; and *(ii)* Request after Accept, request of the new BP is only triggered when the previous one has been accepted. The creation of the sub-contracts inside the enclosing contract at the promise and accept methods respectively solves the issue. The deploying of the sub-contracts must happen at the address returned by the functions.

3 USE CASE: RENT-A-CAR

The first Use Case is well-known in EO. 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, 2020) all four aspect models (*Cooperation Model*, *Action Model*, *Process Model*, and *Fact Model*) are presented. Together they constitute a coherent whole that offers full insight into and overview of the essence of car rental companies.

3.1 Implementation

For Rent-A-Car BPs there are 5 consequent transactions with the *rental completing* being the parent transaction. The *rental completing* transaction can be looked at as the contract between the Rent-A-Car and the Client with conditions that must be fulfilled for the contract to be terminated. These transactions require data sharing between parties as well as trustless control, for this reason, seems fit to implement them resorting to SCs.

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	

The tangling between all the transactions in this Use Case is presented, in Table 2, for easier comparison and an execution overview, resorting only to the Action Rules identified in (DIETZ, 2020) for the Rent-A-Car BP. For the complete code please go to https://github.com/martasaparicio/try.

3.2 Performance

It's difficult to make a technical evaluation of this process as the BC used is the Ropsten test-net. Although Ropsten test-net is the best test-net that reproduces the current production environment, i.e. system and network conditions on the live Ethereum mainnet, because it's Proof-of-Work net, it doesn't allow to change the Block Size for a deeper technological analysis. However, makes available a link to analyze the blocks and their properties. The blocks in this BC have mutable size, being the maximum block size 139 789 bytes and the minimum block size 517 bytes.

Figure 2: Relation between a sample of block size and latency.

Figure 2 shows the expected behavior, for a sample of transactions. The larger the block, the shorter the latency of a transaction. This behavior is as expected due to the nature of the chosen BC. In Ethereum the block size dynamically increases if it starts getting full, and decreases if it starts getting empty. This means that when the block size is greater, the number of transactions submitted is also greater, so there is dynamically an increase in the size of the blocks so that the latency does not increase proportionally.

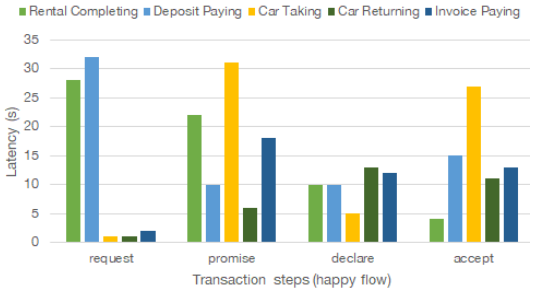

Figure 3: Transaction latency for each DEMO transaction step.

The deployment of the *RentalCompleting* Contract was the operation that took the longest to perform, having a latency of 55 seconds. This behavior is expected as this is the contract that contains all the others.

Regarding the transaction step *request*, 3, the *request* from the *Deposit Paying* transaction was the longest. This may be explained by what was already mentioned. The block size was 2.846 bytes, the smallest found while performing the simulation. This may also be related to the computational capabilities of the miner.

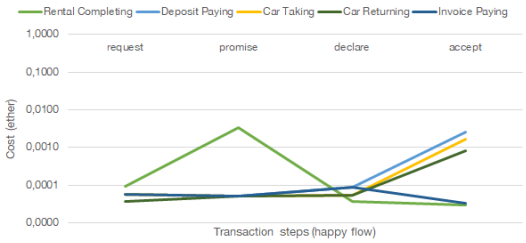

Figure 4: Transaction cost for each DEMO transaction step.

A relevant metric when comparing this solution to others is the overall cost. It is safe to say that the operation that had the greatest financial burden was the deployment of the *RentalCompleting* contract has expected, of 0.004667171 *Ether*, which corresponds to 2.73 $. It was also observed that the price of a transaction is proportional to the complexity of the same.

In Figure 4 it is possible to observe that the most expensive transaction step of the *RentalCompleting* Contract is the promise step where the *DepositPaying* Contract is created. Following the same logic the most expensive step for the contracts *DepositPaying*, *CarTaking* and *CarReturning* is the accept step where the subsequent transactions are created. The process grand total is 0.014027725 *Ether*, which can be converted to 8.20 $.

3.3 Functional Argumentation of Asset Control

To complete a DEMO BP kind, it is necessary to execute a set of BC transactions, meaning a set of BP steps to complete the transaction pattern. For this reason, to complete an inter-organizational scenario, this approach implies the submission of a very large number of transactions. It is necessary to submit, in a happy flow case, a total of 20 transactions, not counting the deployment of 1 contract and four other tacitly, to complete the scenario.

Despite requiring a high number of submitted transactions, this implementation can objectively detect unexpected situations. For instance, an unwanted party tries to take the car that someone else paid for. This is possible due to the use of BC, this technology allows not only to keep track of all transactions but to whom they belong, thus facilitating asset control.

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.

This implementation raises the issue of the disappearance of some transactions that can be tacitly accomplished. Namely, the treatment of sub-transactions should be studied. In this work the the sub-transaction is implemented as another contract, this is not an optimal solution, due to the number of resources that this implies. It is believed there must not always be a need to create separate contract for sub-transactions, the sub-transaction can be implemented inside the main contract. This can be convenient if only partial execution benefits from its BC execution. Finally, the last option is that the sub-transactions are not handled at all and leave this outside of BC. The choice of which option to choose seems at all related to the situation in question. Moreover, it might not be possible to implement full business logic and there is no need to run the exact same transaction execution multiplied on thousands of computers.

4 USE CASE: EUROPEAN UNION PARLIAMENT ELECTIONS

The European Union election process is very complex. The European Union issues general guidelines on how country elections should look like, and each country then implements its legislation to describe how the elections are done in a particular country. This means that each of the 27 European Union countries do this process differently. To avoid this complexity, it was assumed that there is a unified voting process and each country votes according to one of the three voting systems – Preferential Voting, Closed Lists, and Single Transferable Vote.

4.1 Implementation

Let's start this implementation section by understanding the token concept. The use of BC technology seems the future to manage digital assets, due to its security and immutability features. In (Voshmgir, 2019) tokens are described as ways to represent *"programmable assets or access rights, managed by a SC and an underlying distributed ledger. They are accessible only by the person who has the private key for*

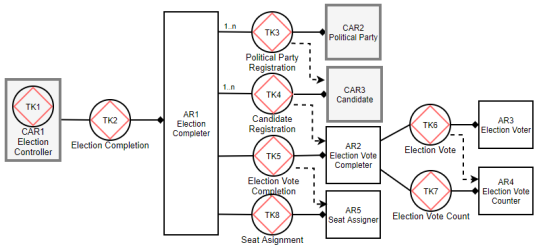

Figure 5: Process Structure Diagram (representation form of the *Process Mode*) of the European Union Parliament Elections process.

that address and can only be signed using this private key" (Voshmgir, 2019).

However, without the use of fungible tokens, this would be impossible to do since no unique information can be written into the token. To represent tokens that have to be unique and hold information instead of values, non-fungible tokens are preferred.

Due to the requirement of using non-fungible tokens for this Use Case, a *VotingToken* contract was created.

```
import "https://github.com/OpenZeppelin/
openzeppelin-contracts/blob/release-v3.1.0/
contracts/access/Ownable.sol";
import "https://github.com/OpenZeppelin/
openzeppelin-contracts/blob/release-v3.1.0/
contracts/token/ERC721/ERC721.sol";

contract VotingToken is Ownable, ERC721{
constructor(string memory name, string memory symbol)
ERC721(name, symbol) Ownable() public payable{}
function mint(address receiver)
onlyOwner public {
_safeMint(receiver, uint256(receiver)); }

function transfer(address from, address to)
onlyOwner public {
_transfer(from, to, uint256(from)); }
}
```

About 30 Action Rules were created to implement the process shown in Figure 5, available at https://github.com/martasaparicio/try. The idea would be for the end result of *ElectionControlling* to result in the deployment of the *ElectionCompleting* SC into the BC.

Although more complex, the chaining of transactions and contracts would process in a similar way to what was exposed in Table 2.

An ontological builder provides the best ontology and knowledge representation practices, together with the best enterprise solutions architecture to provide a robust and scalable ontology management solution. The essential function that is being considered is that an ontological builder translates an ontological model, fully independent of the implementation, into an implementation model, this process being the most straightforward definition of engineering. In software development, the implementation model often is the source code in a programming language like Solidity.

Google Blockly is a visual coding block editor. Blockly documentation claims that this editor is an intuitive, visual way to build code, which is in line with the requirements to model Action Rules. An action rule can be interpreted as a block that contains the normal Action Rules Specification.

To satisfy these requirements it is necessary to create, for each customized block, a Block definition object, a Toolbox reference, and a Generator function. The Block definition object defines the look and behavior of a block, including the text, color, fields, and connections. Once defined,the type name must be used to reference the block to the toolbox. The toolbox reference allows users to add it to the workspace. Finally, to transform the block into code, the block is paired with a generator function. The generator in question is specific to the desired output language, Solidity. The generator function takes a reference to the block for processing. It renders the inputs into code strings and then concatenates those into an expression.

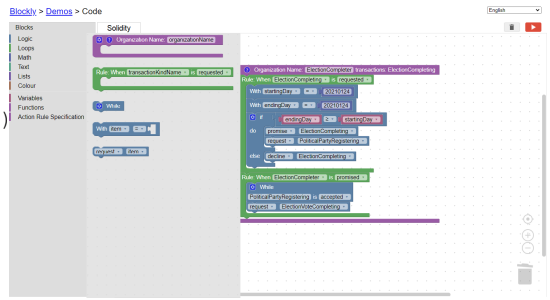

Figure 6: Blockly workspace, defining ARS-5 and ARS-6.

As an example the workspace is presented 6. In the figure the first two action rules to be performed by the actor *Election Completer* are presented.

The Ontological Builder, as already mentioned, was not the main objective of the thesis, so its development still has room for improvement.

4.2 Performance

As proven by 7 figure shows the expected behavior, for a sample of transactions. Both variables have an inversely non-proportional relationship. The block size is not a controllable metric, so it is not a metric by which, can extract knowledge beyond which is public domain about the operation of the BC Ethereum, a lower block size results in a higher throughput.

As previously explained, this use case requires

Figure 7: Relation between a sample of block size and latency.

non-fungible tokens to represent voter's ballots, preventing a double vote situation. This requirement requires additional costs associated with the deployment of the *VotingToken* Contract. The time spent in the deployment of this contract was 14 seconds. The cost was of 0.022150379098 *Ether*, which is equivalent, at the time of writing, to 13,08 $. When compared to the total cost of the Rent-A-Car Use Case of 8.20 $, can be immediately realized that this Use Case has a much higher cost than the previous.

Once again, the conclusions point that he latency of these transactions is predominantly determined by the levels of supply, meaning the miners, and demand in the network.

Without accounting for the cost and time required to create the ballot for each voter, which has already been presented separately, the total cost of this use case was 0.02244436 *Ether*, which is equivalent to 15.43 $, the total time was 654 seconds, which equals 10 minutes and 9 seconds.

4.3 Functional Argumentation of Asset Control

With the application of the BC, it would make sense to implement the *Election Controlling* transaction through an Oracle. An Oracle is a data feed provided by an external service and designed for use in SCs on the BC. In this particular case, the *Election Controlling* transaction would disappear to make way for the implementation of an Oracle, that every five years would trigger the execution of the *Election Completing* transaction.

As the votes are submitted by voters, in this case by the *Election Voter*, their counting is done intrinsically. Which can lead to the disappearance of the *Election Vote Counting* transaction. BC technology has the ability to secure and validate the voting process on its own, as it secures a person's vote and doesn't allow any other actor to change its vote. For this solution, no centralized authority is needed to approve the votes, and everyone agrees on the final tally as they can count the votes themselves, as anyone can verify that no votes were tampered with and no illegitimate votes were inserted.

Now the allocation of seats in parliament can also be done automatically using BC technology. The BC has access to all candidates if the voting system followed is Preferential Voting or Single Transferable, or all candidates associated with a party if the voting system followed is Closed List. When the assignment is being made, it's possible to assign a token to each of them, now deputies. With this token, they would be able to identify themselves in the course of their duties. For example, voting in parliament would be done using this token. This token would allow the citizens of each country to verify the active participation, or not, of the elected candidates.

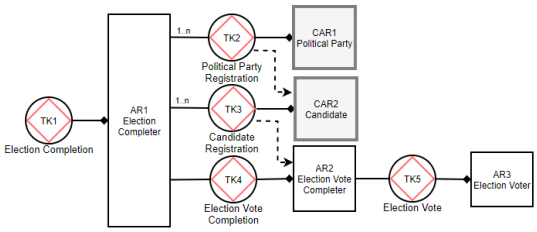

Figure 8: Process Structure Diagram (representation form of the *Process Mode*) of the European Union Parliament Elections process using Blockchain.

The changes would translate into the restructuring shown in 8. As can be seen was made the option of keeping the *Election Vote Completer* actor, for it represents a different real-world entity.

The benefits of applying BC technology to the use case, such as reducing the number of transactions and asset control, have been described. Now, the use of Ontology provides another type of benefit, provide solid and robust principles, in addition to reusing and organizing knowledge.

The limitation of this approach is that mainly there is currently no public BC capable of supporting millions of transactions in a cost and time-efficient way, as it would be necessary for this Use Case.

5 COMMUNICATION

To obtain scientific evaluation about the conceptual mapping developed and presented the following paper was submitted and accepted in:

- **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 (ICEIS)

To obtain scientific evaluation about the Use Case implementation developed and presented the following paper was submitted and accepted in:

- **Aparício, M.**; Guerreiro, S. and Sousa, P. (2020). **Automated DEMO Action Model Implementation using Blockchain Smart Contracts.** In Proceedings of the 12th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (KEOD)

To obtain scientific evaluation about the Use Case implementation developed and presented the following paper was submitted and accepted in:

- Skotnica, M.; **Aparício, M.**; Pergl, R. and Guerreiro, S. (2021). **Process Digitalization using Blockchain – EU Parliament Elections Case Study.** Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)

6 CONCLUSION

This work sets out to address the hypothesis of using BC SCs to implement DEMO *Action Models*. To address the problem set out, this work has presented two artifacts which consist of one conceptual mapping, between DEMO concepts and Solidity concepts, and a base contract, from which all other transaction should inherit. These artifacts aim to remove ambiguities and wrong implementations of the DEMO standard transaction pattern. These two artifacts allow for the development to only focus on the business logic implementation. By using BC technology, it ensures that something that happens cannot be deleted once recorded. Moreover, using Smart Contracts ensures that a given routine is executed every time a transaction, of a given type, is recorded. This alternative method ensures that important business rules are always followed, in addition to facilitating asset control.

The prototype of a possible ontological builder was also presented, although it is not identified as one of the artefacts resulting from the work presented here. By computing a high-level formal model as DEMO *Action Model* and generating Smart Contracts from it, one can create smart contracts that ensure the correctness of transactions data before it is stored in the BC.

Regarding the mapping between the DEMO concepts and Solidity concepts, DEMO ensures the syntactic validation, while the two Use Case ensure the semantic validation. As for the base contract, from which all other transactions should inherit, it instantiates the mapping, and its validation is done using the EVM compiler and Solidity language.

Two distinct use cases were studied. The first,refers to assets that are materialized outside the BC. In this case, the asset (car) studied, is realized outside the BC, but its control is done on the BC. The second, refers to assets that are materialized within the BC. In this case, the asset (vote) studied, is realized within the BC, and its control is done on the BC.

REFERENCES

Andrade, M., Aveiro, D., and Pinto, D. (2018). Demo based dynamic information system modeller and executer. *Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*.

DIETZ, JAN L. G. MULDER, H. B. F. (2020). *ENTERPRISE ONTOLOGY: a human-centric approach to understanding the essence of organisation*. SPRINGER NATURE.

Dietz, J. L. (2006). *What is Enterprise Ontology?* Springer.

Dietz, J. L. (2007). Enterprise ontology-understanding the essence of organizational operation. In *Enterprise Information Systems VII*, pages 19–30. Springer.

Dumas, M., Rosa, M. L., Mendling, J., and Reijers, H. A. (2019). *Fundamentals of Business Process Management*. Springer Berlin.

Markov, K. (2008). Data independence in the multidimensional numbered information spaces. *International Journal*, 2.

Tobias, F. and Jan, R. (2013). Construct redundancy in process modelling grammars: Improving the explanatory power of ontological analysis. *ECIS 2013 - Proceedings of the 21st European Conference on Information Systems*.

van Wingerde, 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. IEEE.

Voshmgir, S. (2019). *Token economy: How blockchains and smart contracts revolutionize the economy*. BlockchainHub.

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.