Towards automatic smart contract generation from DEMO models: A case in logistics using Hyperledger

Leonardo Abreu[1], David Aveiro[1,2,3] and Vítor Freitas[1]

[1]ARDITI - Regional Agency for the Development of Research Technology and Innovation, Funchal, Portugal
[2]University of Madeira, Funchal, Portugal
[3]NOVA - Universidade NOVA de Lisboa, Lisboa, Portugal

Abstract

In this paper, we delve deeper into our prior work which proposed a method for automatically generating smart contracts from business models within the logistics industry, leveraging the Hyperledger Fabric blockchain platform. Building upon our previous contributions, which established a mapping from DEMO (Design and Engineering Methodology for Organizations) language to Hyperledger Chaincode using GO language and extended DEMO's Action Model Grammar, we now present a specific method, transpiling rules and file structure that underpins the automated smart contract generation. This detailed exposition comprehensively explains the flow logic and structural components necessary for translating business rules and processes into executable smart contract code. By presenting the core mechanics of our approach, we aim to offer a foundational tool for enterprises seeking interoperability via distributed ledger technology, further combining the strengths of the DEMO methodology with smart contract functionalities. This research elucidates the nuances of design and implementation and serves as a practical guide for future applications in various business scenarios.

Keywords

Enterprise engineering, DEMO Action Model, Blockchain, Hyperledger Fabric, Smart Contracts, Logistics industry

1. Introduction

Distributed Ledger Technology (DLT) has introduced significant innovations like blockchain (BC), providing a decentralized data system and ensuring consensus among participants [1]. Blockchain technology is valued for its ability to provide an immutable record for decentralized transactions, suitable for financial transactions and also areas like smart contracts, civic services, IoT, and more. It offers businesses unparalleled reliability, transparency, and strong protection against fraudulent activities. Furthermore, it aids in the automation of smart contracts [2].

A Smart Contract (SCs) is a self-executing code on a blockchain platform designed to execute the terms of a contract between parties without intermediaries. The terms are set and recorded on the blockchain, ensuring they are consistently and immutably executed, ideal for parties without an inherent trust basis [3]. SCs utilizing BC technology can transform logistics management by

Companion Proceedings of the 16th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modeling and the 13th Enterprise Design and Engineering Working Conference, November 28 – December 1, 2023, Vienna, Austria
leonardo.tadeu@arditi.pt (L. Abreu); daveiro@staff.uma.pt (D. Aveiro); vitor.freitas@arditi.pt (V. Freitas)
0009-0005-0424-6301 (L. Abreu); 0000-0001-6453-3648 (D. Aveiro); 0009-0002-0667-5749 (V. Freitas)

offering a safe, streamlined, and clear method for monitoring items and dealings, culminating in enhanced supply chain oversight [4].

Hyperledger Fabric is an open-source BC framework that addresses the shortcomings of conventional blockchains. It is implemented in over 400 samples, demonstrations, and operational distributed ledger structures spanning various sectors and applications. Fabric presents a fresh BC design focused on durability, adaptability, expandability, and privacy [5].

The latest advancements in Enterprise Engineering (EE), include using DEMO (Design and Engineering Methodology for Organizations) models for SCs deployments. This highlights the integration of the Construction and Action Models from DEMO with BC technology to ensure safe validation and transaction processing [6, 7].

Utilizing the advantages of BC technology, like immutability and transparency, and combining it with the design and evaluation strengths of the DEMO method, this strategy can result in the optimized and effective application of business procedures. Specifically, this progression holds considerable relevance for crafting and deploying smart contract-driven systems in large-scale business scenarios, bolstering compatibility and growth potential [8].

Even with the latest progress in combining the DEMO approach with SCs deployments, present studies indicate that creating SCs from DEMO models (and designs using other modelling languages) predominantly depends on hand-crafted coding. This scenario restricts the capacity of Distributed Ledger Technologies (DLTs) when implementing DEMO-oriented information systems, turning the contract creation process mainly manual task [9, 10].

While there has been considerable progress in merging the DEMO standard with SCs applications, a pressing need remains to advance the DEMO specification language. The aim is to facilitate the automatic generation of DEMO-based SCs with little to no hand-crafted coding, potentially eradicating manual coding completely. This advancement would boost the capabilities of Distributed Ledger Technologies (DLTs) in deploying DEMO-focused information systems, greatly diminishing manual work and the chance for mistakes in contract development, thanks to early and comprehensive validation by business users and automating repetitive coding tasks.

SCs for Hyperledger Fabric can be automatically generated through the integration of DEMO Action Rule (AR) Models and a Dynamic Search component of a DEMO based Low-Code Platform (DLCP). This approach offers significant advantages, including savings in time and resources, as well as facilitating the automation of data validation and interaction with blockchain data.

In this paper, we will examine a single transaction that involves numerous creations and updates. We will also provide an example illustrating how to retrieve data from the blockchain ledger. To this end, our method introduces two sets of rules, all of which have been employed in a real-life project that confirmed the viability of this auto-generation process.

Research presented in this paper occurred in the context of the LOGHyL (LOGistics HyperLedger project – fictitious name for blind review purposes) initiative which employs blockchain technology to address trust and collaboration issues in logistics. It offers a digital platform using SCs to record micro-hub transactions transparently, allowing stakeholders to access accurate and trustworthy data. This boosts trust and promotes cooperation. Additionally, LOGHyL emphasizes eco-friendly practices through circular economy principles. Overall, LOGHyL, leveraging blockchain and SCs, aims to reshape the Micro-hubs paradigm, promoting a sustainable and efficient system benefiting all involved parties. Micro-hubs are key logistical points

allowing coopetition among carriers, improving service, and reducing costs. However, trust in data exchanges remains a challenge. By integrating key technologies, greener logistics is achievable.

We propose new necessary elements for DEMO Action Model and detail the file structure and actions required to generate GO language methods of a SCs automatically. We can summarize our research question as: How can the DEMO Action Rule Specification (ARS) language be extended to enable automatic generation of SCs, specifically utilizing DEMO AR for transactions and a DLCP Dynamic Search component for retrieving data from the blockchain ledger?

2. Literature review

In the modern global market, there is a pressing need for transparency in supply networks within transportation and production businesses. The demand from consumers and regulators for information about product origins, manufacturing, and movements necessitates robust end-to-end traceability systems. While centralized monitoring has been common, decentralized tracking methods are seen as more trustworthy, secure, and transparent. These methods enhance data protection, reduce inaccuracies, and eliminate single failure points. Despite the technical challenges they present, their benefits outweigh the drawbacks. Adopting such systems allows firms to provide instant data to customers, reduces fraud risks, and ensures regulatory compliance [11].

Open blockchains use digital currency and typically depend on "proof of work" (PoW) for consensus. Conversely, private blockchains involve a set of identified members, ensuring secure transactions among entities with shared goals but limited trust. Fabric, the first blockchain platform of its kind, supports apps written in traditional programming languages. It employs a unique execute-order-validate model, which involves running and checking a transaction, organizing it via consensus and, finally, validating it based on application-specific trust standards [5].

Developing SCs can be intricate and lengthy, necessitating profound knowledge of programming languages and blockchain infrastructures. To address this, Choudhury et al. (2018) suggested a technique utilizing domain-specific ontologies and semantic regulations to automate the conversion of constraints from legal documents or protocols into SCs. This advanced approach employs parsing and abstract syntax tree handling methods to automatically produce SCs from defined requirements, easing the conversion of stipulations from legal contracts or standards, and ensuring repeatability. The method's success was showcased both by clinical study guidelines and vehicle leasing conditions. The suggested approach can potentially be adapted to different programming languages and blockchain systems, making it appealing for enterprises and people wanting to simplify the SCs development process. By automating this development, it becomes faster and more user-friendly, opening up avenues for various entities to leverage the advantages of SCs [9].

Aparicio et al. [6] introduce an innovative approach that employs ontological models to support the effective deployment of SCs within an enterprise's blockchain framework. To illustrate the practical application of their method, the authors delve into a case study centred on the Rent-A-Car business model. Additionally, the research delves into the interplay between

Blockchain-based SCs and the DEMO Action Model, exploring the transformation of one into the other. Central to the paper's findings is the automated knowledge extraction from the DEMO Action Model, which aids generating Blockchain SCs.

Other previous work [7] discussed the usage of blockchain technology to streamline processes between mistrusting organizations. Their study underscores the use of SCs in DEMO Action Models and explores model-driven engineering for automatically generating these contracts. The paper advocates for ontology-centric modelling techniques to ensure precise SCs specifications. While a connection between Solidity Concepts and DEMO Action Model concepts is presented, its optimization remains a topic for future research. The study uses a Rent-A-Car scenario for validation but suggests broader testing across different Action Models. In essence, ontology-driven approaches can enhance data standards, business processes, and blockchain management, particularly in SCs, leading to better understanding and increased blockchain security.

Tong et al. (2022) [12] introduced an innovative approach to automate the creation of SCs. Their framework, termed AI-assisted SCs Generation (AIASCG), facilitates collaboration and negotiation on contract clauses among contracting parties from varied backgrounds and languages. The method offers a universal portrayal of contracts using machine natural language (MNL), ensuring a mutual comprehension of contract commitments. Key to this proposal is an AI-driven technique for word segmentation named Separation Inference (SpIn). SpIn is pivotal to AIASCG, efficiently translating natural language sentences into intermediate MNL forms, greatly minimizing the manual intervention in generating contracts.

In Tallyn et al.'s study [13], the application of SCs in four urban delivery models was examined, highlighting the balance between automation and the human touch necessary to maintain trust during deliveries. The research underscores the importance of interpersonal relationships in ensuring trust between delivery personnel and recipients.

Kim and Laskowsky et al. [14] used the BC structure from the TOVE traceability ontology to develop SCs for tracking physical goods in global supply chains. These contracts adhere to traceability regulations based on ontology, emphasizing the ability of SCs to trace assets throughout complex supply chains.

We find a gap in the literature regarding practical solutions for the specific logistical needs of circular economy procedures using blockchain in collaborative micro-hubs. A scalable framework is needed to fully utilize this approach, enabling the creation and application of SCs for all stakeholders in a cooperative and user-friendly way.

Even with the advancements reported in the domain of SCs, there remain significant barriers that preclude non-technical individuals from easily engaging in the SCs creation process. The vast majority of frameworks and structures employed are complex and often difficult to comprehend for the untrained eye. A notable observation from the literature is that the methodology for elucidating and deploying these contracts largely relies on text-based explanations. This particular mode of presentation can prove to be particularly daunting and perplexing for individuals lacking technical expertise, highlighting a prominent discrepancy in the existing literature.

Our proposition centers on breaking down the barriers surrounding SCs generation through the integration of visual component. By converting the traditionally text-heavy process into a visual specification, we anticipate that users can more intuitively grasp the essence and framework of the SCs. This visually-driven methodology doesn't just demystify the complexities

but also fosters confidence among users. As they can see and interact with the visual elements, they become more engaged in both the creation and comprehension of the SCs. Our emphasis on visual components seeks to not only simplify but also democratize the SCs landscape, inviting participation from a broader spectrum of individuals, irrespective of their technical prowess.

3. Bridging DEMO with GO smart contracts

In this section, we elaborate on the procedure of generating Hyperledger Chaincode Go from AR Models and DLCP Dynamic Search. Given the extensive nature of the implementation, we provide a link [15] that contains the relevant code, facilitating a thorough examination. Before delving in our most current contributions, we provide some context and summary of prior research work in this line.

3.1. Research context

The goal of implementing information systems is to enhance organizational efficiency. However, success hinges on elements like system functionalities and organization's attributes. DEMO employs the PSI theory of Enterprise Ontology to dissect the core of an organization, forming cohesive models and diagrams that mirror reality accurately [16]. We introduce new language elements for SCs creation from DEMO models, aiming for higher efficiency and reduced errors.

In recent work/studies we extended and validated DEMO's Models with improvements on the representations of it is Fact, Process and Action Models [17, 18, 19, 20]. Regarding the Action Rule Specification (ARS) language, a comprehensive and explicit specification of logical and mathematical expressions, the specification of complex business rule logic and flow became feasible [16]. Such meticulousness is paramount in SCs, where minor mistakes can lead to substantial repercussions.

Furthermore, our proposal appeals to a broader spectrum by incorporating rules via a visual programming language. This facilitates involvement from individuals with limited technical backgrounds, enhancing the entire developmental phase. Our approach paves the way for future SCs projects on the blockchain, by harnessing this synergistic blending of formal language and visual programming.

This paper extends prior work presented in [21], where we focused on the development of an initial prototype that gave us a valued initial insight on how to transpile DEMO ARS to SCs and also helped us with an initial understand of what blocky components and grammar extension we should need to successfully automatically generate SCs within the logistics industry context, using the Hyperledger Fabric as our blockchain platform.

That previous research gave us a good starting perspective for this paper, since that work lead us directly in the direction of gaps that need to be tackled in the grammar specification that need to be adjusted for specifics blockchain needs.

In the previous work, we also demonstrated that DEMO Fact Model can be used for creating asset models that will be stored on the blockchain.

The study presented in this paper explores two specific scenarios derived from the "LOGHyL" project. Firstly we are going to look at auction creation, where we will explore assets creation and updates, each coupled with their relevant verifications. The second scenario that we will

look for is how can we fetch auctions from the ledger based on is state. The previous mentioned scenarios will be categorically labelled as: “T01 – Create Auction” and “T02 – Fetch Open Auctions”. These examples elucidate processes to asset creation and modification, while all the validations are strictly validated, in the T01, furthermore T02 it is a good example of query ledger.

3.2. EBNF grammar specification

In the realm of computer science, Extended Backus–Naur Form (EBNF) constitutes a spectrum of meta-syntax languages, each variant capable of representing a context-free grammar. The primary application of EBNF lies in its utility for the formal articulation of formal languages, inclusive of programming languages utilized in computing.

The subsequent section delineates tables that elucidate the syntax employed in the formulation of Action Rules. The table is structured in the following manner: the left column enumerates the concepts requiring definition, where a majority of these concepts correspond to specific blocks in the system, and others relate to selectable fields within these blocks. Conversely, the right column of the table is dedicated to providing the explicit definitions of these concepts.

Additionally, this section introduces new components of our EBNF-based [22]. The complete grammar, along with further details of these new components, will be presented in the appendices, providing a comprehensive overview of the system’s syntax and its extensions."

Before analyzing the upcoming tables that illustrate the grammar implemented by Extended Backus–Naur Form (EBNF), it is important to acknowledge keynotes on this grammar:

- “( )” means grouping
- “[ ]” is for optional elements (zero or one time)
- “ ” is for optional elements (zero or more times)
- “|” means alternative
- “-” means Syntactic Exception. Separates a rule which must be used (on the left) from the rule describing what is not allowed to be used (on the right) (“Consonant = Letter - Vowel;”). If there’s nothing on the right, it can be interpreted only as something that must be used (“OneOrMore = Something-;”).
- elements with UPPERCASE mean a terminal symbol with specific behavior assigned to the system/dashboard.

Table 1: DISME’s Complete Action Rule’s EBNF Syntax

when	WHEN transaction_type IS | **HAS_BEEN** transaction_state { action}-
local_endpoint_parameter	STRING property documentation_description parameter_example_value validation_condition [MANDATORY] NOTE: the parameter name that will be used to specify the parameter is independent of the internal property that is linked to it, and therefore doesn’t have to be the equal to its name.

local_endpoint_request_body	{local_endpoint_parameter | local_endpoint_parameter_set}- NOTE: defines the structure of the API Call's request_body, defining its parameters and/or parameter_sets.
local_endpoint_parameter_set	STRING documentation_description {local_endpoint_parameter}- NOTE: After specifying the set_name, it is needed to specify the parameters that are to be inserted into it.
local_endpoint_response	local_endpoint_success_response local_endpoint_error_response NOTE: must have a behaviour selected for each one. One defined in case the API call is carried out successfully, and one in case it encounters any errors in the process.
local_endpoint_success_response	local_endpoint_success_response_affected_object | local_endpoint_success_response_queried_object | STRING NOTE: When successful, it can return the created/queried/updated/deleted object or simply a custom success message. NOTE: API Call CRUD operation that will be carried influences the dropdown choices that will be present in the 'response' block. For 'create', we will have the 'created object', for 'read' we will have the local_endpoint_success_response_queried_object, for 'update' we will have the 'updated object', and for 'delete' we will have the 'deleted object'. Except for the 'read' crud operation, we have the option to output a custom success message instead of the object.
local_endpoint_success_response_affected_object	STRING {response_property | response_set}- NOTE: defines the returning object with the object's name and the properties/property_sets that should be returned inside it.
response_property	STRING property NOTE: has to be a property belonging to the object that was created/updated/queried/deleted
response_set	STRING {response_property}- NOTE: defines the set name and the properties that should be included in this response set.
local_endpoint_success_response_queried_object	STRING NOTE: the variable names defined in the 'query records' blocks will appear here in a dropdown and the user must select which ones to include in the api call's response. Will return the objects from the selected queries.
local_endpoint_error_response	STRING NOTE: error message to be returned.
action	causal_link | assign_expression | user_input | edit_entity_instance | user_output | produce_doc | if | while | for_each | post | get | create_record | query_records | update_record | delete_record
assign_expression	property "=" (term | property_value) [BLOCKCHAIN_EXECUTION]

property	STRING NOTE: has to be an existent property specified in table property
validation_condition	[NOT] validation_condition_type [EXTRA_FIELD_1 [EXTRA_FIELD_2]] [user_output] NOTE: The not, extrafield1 and extrafield2 fields will appear depending on the validation_condition_type chosen. EXTRAFIELD1 and EXTRAFIELD2 examples are the min and max fields if the validation_condition_type is "Belongs Range".
validation_condition_type	REQUIRED | IS_NUMBER | IS_INTEGER | EQUAL_TO | MAX_WORD_LENGTH | LESS_EQUAL | HIGHER_EQUAL | HIGHER_THAN | LESS_THAN | MIN_LENGTH | BELONG_SRANGE | MAX_LENGTH | MIN_WORD_LENGTH | HAS_CHARACTER | REG_EXPRESSION | HAS_WORD | IS_EMAIL | IS_URL | CUSTOM_VALIDATION
compute_expression	term {compute_operator term}-
compute_operator	"+"| "-"|"*"|"/" NOTE: for now simple mathematical operators, later maybe more...
if	IF condition THEN { action | rollback_transaction } - [ELSE { action | rollback_transaction } -]
condition	(ISTRUE | NOT evaluated_expression | condition) | (AND|OR { evaluated_expression | condition }-) NOTE: if condition is of type ISTRUE engine will evaluate the expression; if condition is of type NOT, engine will evaluate either the expression or condition; if it is of type AND or OR it will accordingly evaluate the specified expressions/conditions
evaluated_expression	comp_evaluated_expression | user_evaluated_expression
comp_evaluated_expression	term logical_operator (term | property_value)
logical_operator	"<" | ">" | "==" | "!="
property_value	STRING NOTE: must be a possible value of a property with 2 cases: 1) value_type is enum and one can select all allowed values for the selected property 2) value-type is prop_ref where possible values to select will be the values associated with the property fk_property_id.
term	constant | value | property | query | compute_expression | produce_doc
constant	STRING NOTE: Can be a constant defined on the system or can be a new specification of a constant. In the latter case, the name, value and value type of the constant to be created must also be specified.

value	value_type STRING NOTE: free value inserted when editing the action rule. Must also specify the value type when specifying the new value.
value_type	TEXT | INTEGER_NUMBER | REAL_NUMBER | BOOLEAN | ENUM | DATE | TIME
for_each	FOR_EACH set {action}-
set	STRING NOTE: set of elements. Has to be a name of a 'parameter set' defined in the action rule's 'request body' input.
create_record	entity_type {matching_property}- [ALLOW_DUPLICATES] [BLOCKCHAIN_EXECUTION] NOTE: Allows the creation of an entity in the system. 'Allow duplicates' is a checkbox that will/won't allow the insertion of duplicate records of entities of the entity type selected.
entity_type	STRING NOTE: has to be an existent entity_type specified in table ent_type
update_record	entity_type matching_id_property matching_property- [BLOCKCHAIN_EXECUTION] NOTE: properties to be updated in the entity are the matching_properties inserted. The matching_id_property is to know which entity to update.
delete_record	entity_type matching_id_property [BLOCKCHAIN_EXECUTION] NOTE: will search for the selected entity type's entity with the received api call id and will delete that entity.
rollback_transaction	STRING NOTE: does a rollback on the current action rule's transaction being executed and returns the custom error message defined. NOTE: For now, rollback_transaction is only used in blockchain action rules, but in the future will be reused for general action rules.

3.3. From DEMO modelling to Chaincode implementation: Ledger storage

In this section, we will delve into the process of transpiling DEMO Action Models to blockchain transactions encoded in SCs, encompassing multiple creations and updates. To illustrate this, we will examine the creation of an auction, shedding light on the significance of each action within the DEMO ARS.

In the "LOGHyL" project, creating an auction involves conducting multiple validations and establishing a relationship between a parcel and the auction itself. Specifically, this process necessitates the creation of the "Auction_Has_Parcel" entity. This entity acts as a pivot table, managing the many-to-many relationship between Auction and Parcel entities. Additionally, it is imperative to alter the state of the parcel from "Pending" to "Auction", indicating that the parcel is part of an ongoing auction. Prior to making this update, a crucial validation step must be taken to ensure that the parcel's current state is indeed "Pending".

Now, we will dissect the AR into smaller segments to gain a deeper understanding of the significance of each component within the AR, and to comprehend how it is transpiled into Chaincode.

The foundation of the AR is encapsulated in the "When" block, as illustrated in Fig. 1. This segment explicitly outlines various critical components: the AR name, the parameters required for the blockchain request, and the series of actions slated for execution. Additionally, it defines the response mechanism, indicating whether the action has been successfully executed or if any errors have emerged during the process.

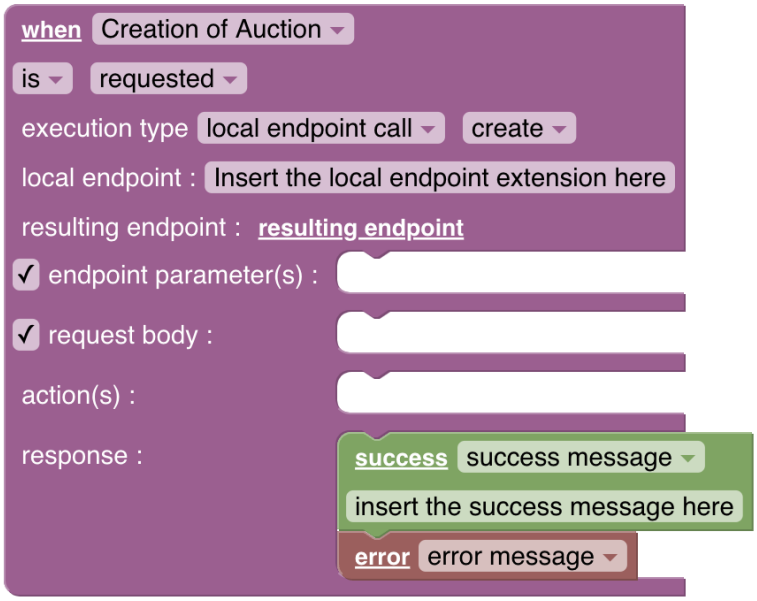

Figure 1: DEMO Action Rule: "When" block

In Fig. 1 has said previously, we have a property that handles with the parameters that our blockchain method have to receive in order to use it in the actions. We can have two types of parameters:

The "parameter" block symbolizes a singular value, as depicted in Fig. 2. The former illustrates a parameter specification devoid of validation, while the latter demonstrates the inclusion of validation for the corresponding parameter.

The "parameter set" represents the second type of parameter, and it denotes a list of parameters sharing the same structure. Utilizing this format streamlines the process of conveying repetitive information, eliminating the need for multiple insertions of the same data type, thereby enhancing intuitiveness and efficiency. Furthermore, validations can be incorporated since the "set parameter" accommodates individual "parameter" as its properties, allowing for comprehensive data integrity checks.

Now, we will examine the Chaincode generated based on the "When", "parameter" and "set parameter" blocks. In the auction example, we encounter multiple "parameter" blocks associated with the auction's properties, as well as a "set parameter" block representing the parcels to be

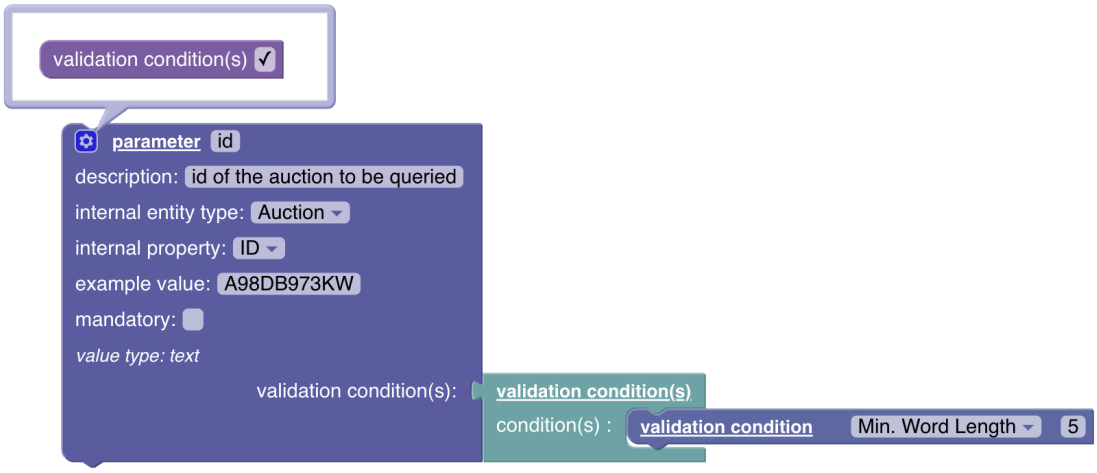

Figure 2: DEMO Action Rule: Parameter Block with validation

included in the auction. The "When" block generates the fundamental structure of the method, taking the aforementioned parameters as input for the base method.

Listing 1: Pseudo Code of Creating Auction Main Method

```
func (s *SmartContract) AuctionCreateMain(stub shim.ChaincodeStubInterface,
    auction models.Auction, parcels []string) pb.Response {
   pseudoInitiateTransactionEnsureAtomicity()
   //ADD ACTIONS HERE
   (...)
   return shim.Success(nil)
}
```

In order to guarantee atomicity, we used a "StartTransaction" method provided by Stub Chaincode Interface, it handles potential errors during the transaction, and ensures the transaction is either rolled back if there were errors or committed if the transaction was successful.

With the foundational method established, we can delve deeper into incorporating actions within the DEMO AR, specifically for executing data writes on the ledger. To facilitate this, we will introduce and discuss two blocks capable of writing data to the ledger: "create record" and "assign expression".

As illustrated in Fig. 3, the "create record" block possesses several properties. The "entity type" property defines the type of entity to be created on the ledger. The "allow duplicates" property can be enabled if specific situations warrant the creation of duplicate records. Lastly, the "property(ies)" property assigns values to various attributes of the auction. By employing the "matching" block, one can link auction properties to parameters that were previously passed to the AR under analysis, ensuring proper initialization and consistency in the data recorded on the ledger.

Now that we have all the necessary information that is needed in the block of creation, we can check the Chaincode that will be generated every time we need to create a record on the

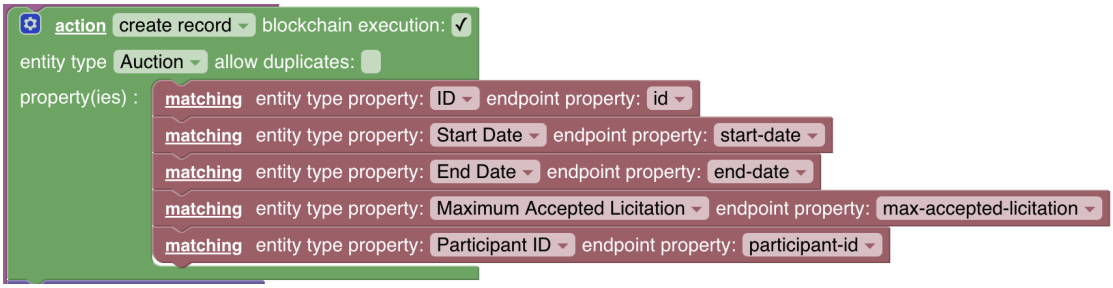

Figure 3: DEMO Action Rule: Create Auction

ledger.

Listing 2: Pseudo Code of Creating Auction

```
func (s *SmartContract) AuctionCreate(stub shim.ChaincodeStubInterface,
    auction models.Auction) error {
  pseudoCreateCompositeKey()
  pseudoValidateAuction()
  pseudoCheckDuplicates()
  pseudoSetDocType()

  bytes, err := json.Marshal(auction)
  if err != nil {
    return err
  }

  _, err = s.UpsertEntityRecord(stub, compositeKey, bytes)
  if err != nil {
    return err
  }

  return nil
}
```

It is observable that the auction is passed as an argument to the "AuctionCreate" function, and there are consistently steps followed for each creation on the ledger.

Firstly, a unique key is generated to create a composite key, which facilitates the identification and retrieval of the record. Following this, the properties of the entity are validated. This validation process is generated automatically, based on any validations specified in the "parameter" component. After validation, a check is performed to verify if duplicate values are permissible for this particular record. Subsequently, a document type (docType) property is assigned to the entity, which aids in the categorization and querying of ledger records. Lastly, the entity, along with all its validated properties and the assigned docType, is securely stored on the ledger.

To invoke this method from the main function, a specific code block needs to be integrated into the "AuctionCreateMain" method. Here is how it will appear once this action has been

implemented:

Listing 3: Pseudo Code of Creating Auction Main Method with create auction added

```
err = s.AuctionCreate(stub, auction)
if err != nil {
   transactionError = true
   return shim.Error(err.Error())
}
```

In order to create the records for the entity that is responsible for the management of many-to-many relation between auction and parcel, e need to add a new component to it the “for each” block, this receives a set of parcels in this case and creates for each parcel a record of type “Auction Has Parcel”, as showed in Fig. 4.

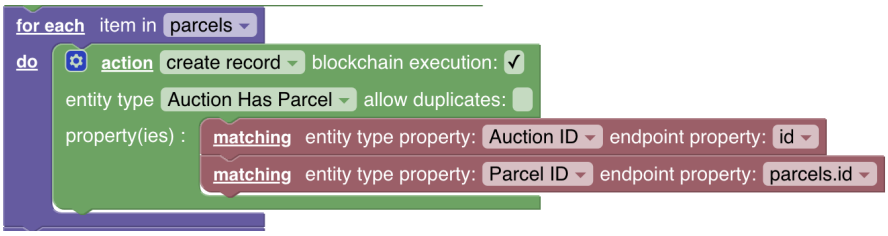

Figure 4: DEMO Action Rule: Create Auction Has Parcel

The process of creating an ’auction’ entity and an ’auction has parcel’ entity differs in only one specific aspect. This distinction lies in the adaptation of the generated function within the ’for each’ loop to accommodate the ’Auction_Has_Parcel’ entity type. Specifically, in the ’AuctionCreateMain’ section under actions, an additional ’for each’ loop is incorporated. The code snippet added to this loop is as follows:

Listing 4: Pseudo Code of Creating Auction Main Method with create auction has parcel added

```
for _, parcelId := range parcels {
   err = s.AuctionHasParcelCreate(stub, auction.ID, parcelId)
   if err != nil {
      transactionError = true
      return shim.Error(err.Error())
   }
}
```

To generate records for the entity that manages the many-to-many relationship between auctions and parcels, it is necessary to incorporate the “for each” block. This block accepts a set of parcels, and for each parcel, it creates a record of the type “Auction Has Parcel.” This process is visually depicted in Fig. 4, illustrating how the “for each” block facilitates the efficient creation of multiple relational records between auctions and parcels.

The “assign expression” component is utilized for updating properties in existing records, and it comprises several key properties. The “Scope” property determines the type of object to be updated; in the example depicted in Fig. 5, the intention is to update an Entity record. The

“ent type” is specified as “Parcel”, indicating the entity type to be updated, and the “property” is set to “State”, pinpointing the specific parcel property to be modified during the update operation. On the right side of the equal sign, the desired value to be assigned to the updated property is clearly presented. This configuration ensures precise and intentional updates to record properties.

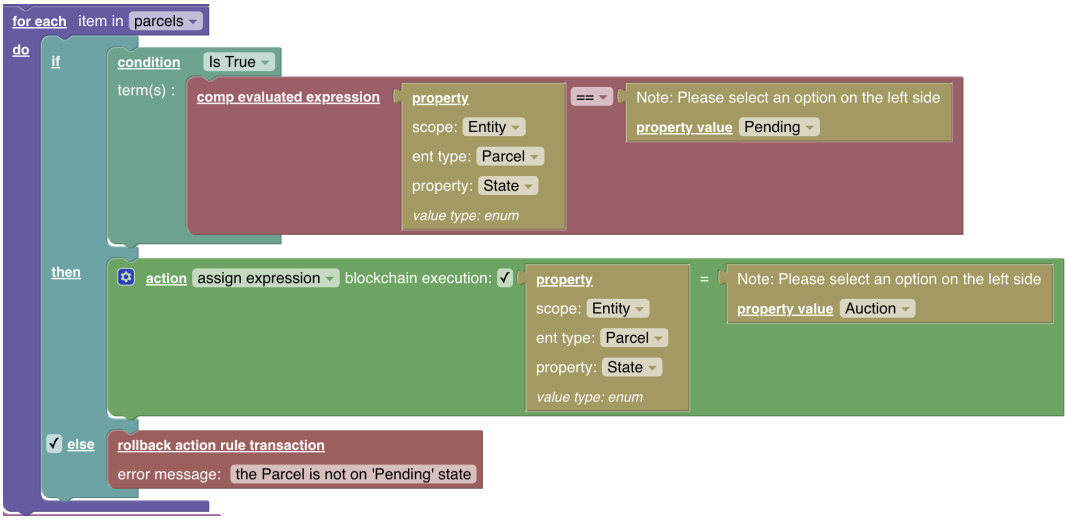

Figure 5: DEMO Action Rule: Update Parcel State

Looking carefully at the component showed in Fig. 5, we can use all the information in that component to perform the update using Chaincode, for that we will need to create the method “ParcelAuctionStartUpdateState” with the following structure:

Listing 5: Code of Updating Parcel State

```
func (s *SmartContract) ParcelAuctionStartUpdateState(stub
    shim.ChaincodeStubInterface, parcelId string) error {
   pseudoCreateCompositeKey()
   pseudoReadEntity(COMPOSITE_KEY)
   pseudoValidateParcelState(PARCEL_STATE)

   parcel.State = models.State(models.ParcelStateAuction)
   bytes, err := json.Marshal(parcel)
   if err != nil {
      return err
   }
   _, err = s.UpsertEntityRecord(stub, compositeKey, bytes)
   if err != nil {
      return err
   }
   return nil
}
```

3.4. From DEMO modelling to Chaincode implementation: Ledger reading

The DLCP Dynamic Search component is a recent innovation in our DLCP, streamlining the process of crafting queries to visualize data from various system entities. The user begins by selecting the desired entity or entities through select boxes, defining the scope of their query. Following this, they choose the specific properties they want to retrieve and designate certain properties to serve as filters in the subsequent stage. In the final step, the user applies these filters based on the properties they selected earlier. This intuitive three-step process enables users to effortlessly design data queries that suit their preferences, all while requiring no technical expertise.

To better understand the DLCP Dynamic Search component, let's explore an example where a user wants to query all open auctions in the system. Initially, the user selects the "Auction" entity. Following that, in the property selection phase, the "state" property is chosen as a filter. Finally, the user specifies a filter condition, setting the "state" to "OPEN". After saving these preferences, a corresponding method is auto-generated in the SCs, adhering to a predefined pattern to ensure consistent and accurate data retrieval.

From the example provided earlier, and based on the information supplied by the user, we can derive the query: {"selector": {"docType": "AUCTION", "state": "OPEN"}}. Utilizing this data, we can translate the user-created input from the DLCP Dynamic Search into the corresponding Chaincode, ensuring seamless interaction and data retrieval from the blockchain ledger.

Listing 6: Query Open Auction

```
queryString := `
  {"selector":
    "{\"docType\":\"AUCTION\", \"state\":\"OPEN\"}
  "}`
resultsIterator, err := stub.GetQueryResult(queryString)
```

As data accumulates within the ledger, retrieval operations can become increasingly time-consuming. To mitigate this latency, Hyperledger Fabric introduces the concept of the "State Database", the world state maintains the current attribute values of a business object as a distinct ledger status. This is beneficial since applications typically need the present value of an object. Traversing the entire blockchain to determine an object's current value would be inefficient, instead we can directly retrieve it from the "State Database". While the "State Database" provides a structured snapshot of the ledger's information, search operations can still experience latency, as data continues to amass. To counteract this, custom indexes are tailored for each distinct query executed on the ledger. Specifically, for the "Query Open Auctions" operation, the following index was automatically generated:

Listing 7: Designing an Index to Access Open Auctions

```
{
  "index": {"fields": [ "docType", "state" ]},
  "name": "auctionByStateIndex",
  "type": "json"
}
```

This analysis demonstrates how data queries can be designed in the DLCP Dynamic Search component, which can subsequently be seamlessly converted into Chaincode. Additionally, performance considerations were discussed in the context of ledger searches. Enhancements were made by adding indices to the Hyperledger's "State Database", optimizing the search queries crafted for the SC.

3.5. Structure of generated SC files

In the endeavour to automate SC generation, it was imperative to conceive an intuitive method to structure the output files resulting from the transformation of DEMO AR to Chaincode. For the proposed structure (Fig. 6), each AR will generate a corresponding file within the "Chaincode" directory. Simultaneously, individual entities will have their data models encapsulated in distinct files located in the "Models" directory.

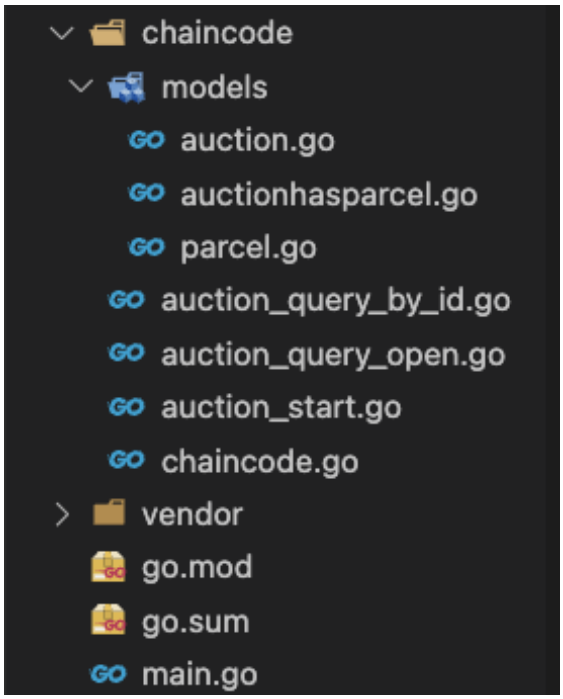

Figure 6: Structure of Generated Files

This methodology is employed because users create each DEMO AR individually. If an error occurs or a modification is necessary in a specific AR, it is more straightforward to pinpoint and remove the generated file associated with that particular AR. Subsequently, a new file can be created based on the alterations made to the AR. This process enhances the ease of error correction and adjustment, leading to a more efficient and user-friendly experience.

4. Results and discussion

The findings of this research illustrate the feasibility of automatic SC generation for Hyperledger Fabric using DEMO AR models and DLCP Dynamic Search. Such an approach offers a multitude of advantages, including enhanced efficiency in the creation process for intricate contracts, thereby reducing both the time and resources required. Furthermore, it provides capabilities for automated data validation, error handling, and interaction management within the blockchain.

Through a detailed study of a specific transaction "Creation of Auction" and method to "Query Open Auctions" we developed a SC in Go. Subsequent reverse engineering exercises further reinforced the viability of automatic SC generation. Our exploration validated various automatic generation scenarios encompassing aspects like data validation, relationships between system entities, iterative processes among new transactions and existing blockchain data, sophisticated blockchain queries, and robust error management.

The AR models presented in figures above include many new elements for DEMO's ARS Language. For time and space reasons, we add these extensions to EBNF in the link [15].

The insights garnered from this study suggest that leveraging the DEMO AR models can be a wise strategy for automatic SC generation development. It streamlines the conception of intricate contracts and provides insights into the organization of the resultant files.

However, potential limitations should be acknowledged. The inherent expressiveness of the DEMO language may constrain the crafting of more advanced contract logic. This realization highlights the potential need to augment the DEMO language components to address blockchain-centric challenges. By integrating these refinements, the proposed framework can evolve into a holistic solution, thereby facilitating the development of increasingly intricate and versatile SC within the blockchain.

5. Conclusions and future work

This research showcases the promising potential of utilizing DEMO AR models for the automated generation of SC, potentially saving significant time and resources in developing complex contracts. This approach offers automation in critical areas, such as data validation and error management, and also ensures robust data interaction within the Blockchain. Additionally, the study delves into aspects of performance in blockchain queries and proposes a structured format for the files that will be generated automatically.

Furthermore, the adaptability of this approach suggests applications across various business sectors, provided there is an appropriate DEMO model to depict the business process. Such versatility could spearhead innovations in SC development, possibly leading to broader blockchain technology adoption.

Regarding future work, one research line could be the integration of advanced functionalities into automatically generated contracts. For instance, implementing more sophisticated filtering mechanisms at the level of queries can ensure that blockchain data searches are conducted more transparently Instead of merely returning a single entity, the system could retrieve related entities, thereby providing a richer dataset. DLCP Dynamic Search holds significant potential for aiding future developments in this area of research. As we increase this complexity,

it is imperative to consider performance aspects. Enhancing the system's efficiency, while maintaining its robustness, is crucial to ensure that users get timely and relevant information without compromising speed and reliability, in the future, we aim to develop an optimized method for creating indexes based on the queries specified through the DLCP Dynamic Search component. Though DEMO models are adaptable, specific sectors might necessitate nuanced grammatical rules or modifications. This implies that while our current foundational structure of DEMO models for SCs can serve a broad range of subjects, there might be a need for further tailored adjustments or refinements to cater to the unique requirements of blockchain.

An innovative low-code platform grounded in DEMO is underway, aiming to transpile DEMO models directly into GO code, simplifying blockchain implementation, facilitating the architecture of involved entities and automating essential API code generation.

The DEMO AR models could be pivotal in SC evolution, especially within the Hyperledger Fabric environment. The merits of this methodology are evident, and further research should broaden the capabilities of generated contracts and evaluate optimized approaches tailored to each specific scenario.

References

[1] S. Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System, Cryptography Mailing list at https://metzdowd.com (2009).

[2] Z. Zheng, S. Xie, H. Dai, X. Chen, H. Wang, An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends, in: 2017 IEEE International Congress on Big Data (BigData Congress), 2017, pp. 557–564. doi:10.1109/BigDataCongress.2017.85.

[3] N. Szabo, Smart Contracts : Building Blocks for Digital Markets, 2018. URL: https://www.semanticscholar.org/paper/Smart-Contracts-%3A-Building-Blocks-for-Digital-Szabo/9b6cd3fe0bf5455dd44ea31422d015b003b5568f.

[4] N. Álvarez Díaz, J. Herrera-Joancomartí, P. Caballero-Gil, Smart contracts based on blockchain for logistics management, in: Proceedings of the 1st International Conference on Internet of Things and Machine Learning, IML '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 1–8. URL: https://dl.acm.org/doi/10.1145/3109761.3158384. doi:10.1145/3109761.3158384.

[5] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, J. Yellick, Hyperledger fabric: a distributed operating system for permissioned blockchains, in: Proceedings of the Thirteenth EuroSys Conference, EuroSys '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 1–15. URL: https://dl.acm.org/doi/10.1145/3190508.3190538. doi:10.1145/3190508.3190538.

[6] M. Aparício, S. Guerreiro, P. Sousa, Automated DEMO Action Model Implementation using Blockchain Smart Contracts, 2020. doi:10.5220/0010147602830290.

[7] M. Aparício, S. Guerreiro, P. Sousa, Decentralized Enforcement of DEMO Action Rules Using Blockchain Smart Contracts, 2021.

[8] B. Hornáčková, M. Skotnica, R. Pergl, Exploring a Role of Blockchain Smart Contracts

in Enterprise Engineering: 8th Enterprise Engineering Working Conference, EEWC 2018, Luxembourg, Luxembourg, May 28 – June 1, 2018, Proceedings, 2019, pp. 113–127. doi:10.1007/978-3-030-06097-8_7.

[9] O. Choudhury, N. Rudolph, I. Sylla, N. Fairoza, A. Das, Auto-Generation of Smart Contracts from Domain-Specific Ontologies and Semantic Rules, 2018. doi:10.1109/Cybermatics_2018.2018.00183.

[10] M. Skotnica, R. Pergl, Das Contract - A Visual Domain Specific Language for Modeling Blockchain Smart Contracts, 2020, pp. 149–166. doi:10.1007/978-3-030-37933-9_10.

[11] E. Tijan, S. Aksentijevic, K. Ivanić, M. Jardas, Blockchain Technology Implementation in Logistics, Sustainability 11 (2019) 1185. doi:10.3390/su11041185.

[12] Y. Tong, W. Tan, J. Guo, B. Shen, P. Qin, S. Zhuo, Smart Contract Generation Assisted by AI-Based Word Segmentation, Applied Sciences 12 (2022) 4773. URL: https://www.mdpi.com/2076-3417/12/9/4773. doi:10.3390/app12094773, number: 9 Publisher: Multidisciplinary Digital Publishing Institute.

[13] E. Tallyn, J. Revans, E. Morgan, K. Fisken, D. Murray-Rust, Enacting the Last Mile: Experiences of Smart Contracts in Courier Deliveries, 2021, pp. 1–14. doi:10.1145/3411764.3445525.

[14] H. Kim, M. Laskowski, Toward an ontology-driven blockchain design for supply-chain provenance, Intelligent Systems in Accounting, Finance and Management 25 (2018) 18–27. doi:10.1002/isaf.1424.

[15] Towards automatic smart contract generation from demo models: a case in logistics using hyperledger, 2023. Url:https://bit.ly/loghl original-date: 2023-09-25.

[16] D. Aveiro, J. Oliveira, Towards DEMO model-based automatic generation of smart contracts, in: e. a. Cristine Griffo (Ed.), Advances in Enterprise Engineering XVI - 12th Enterprise Engineering Working Conference, EEWC 2022, Leusden, The Netherlands, November 2-3, 2022, volume 473 of *LNBIP*, Springer, 2022, pp. 71–89. URL: https://doi.org/10.1007/978-3-031-34175-5_5. doi:10.1007/978-3-031-34175-5_5.

[17] D. Pacheco, D. Aveiro, D. Pinto, B. Gouveia, Towards the x-theory: An evaluation of the perceived quality and functionality of demo's process model, in: D. Aveiro, H. A. Proper, S. Guerreiro, M. de Vries (Eds.), Advances in Enterprise Engineering XV, Springer International Publishing, Cham, 2022, pp. 129–148.

[18] D. Pacheco, D. Aveiro, B. Gouveia, D. Pinto, Evaluation of the perceived quality and functionality of fact model diagrams in demo, in: D. Aveiro, H. A. Proper, S. Guerreiro, M. de Vries (Eds.), Advances in Enterprise Engineering XV, Springer International Publishing, Cham, 2022, pp. 114–128.

[19] D. Pinto, D. Aveiro, D. Pacheco, B. Gouveia, D. Gouveia, Validation of demo's conciseness quality and proposal of improvements to the process model, in: D. Aveiro, G. Guizzardi, R. Pergl, H. A. Proper (Eds.), Advances in Enterprise Engineering XIV, Springer International Publishing, Cham, 2021, pp. 133–152.

[20] B. Gouveia, D. Aveiro, D. Pacheco, D. Pinto, D. Gouveia, Fact model in demo - urban law case and proposal of representation improvements, in: D. Aveiro, G. Guizzardi, R. Pergl, H. A. Proper (Eds.), Advances in Enterprise Engineering XIV, Springer International Publishing, Cham, 2021, pp. 173–190.

[21] D. P. David Aveiro, Leonardo Abreu, V. Freitas, Demo models based automatic smart

contract generation: a case in logistics using hyperledger, 2023. URL: https://doi.org/10.62036/ISD.2023.18. doi:https://doi.org/10.62036/ISD.2023.18.

[22] D. Aveiro, V. Freitas, A new action meta-model and grammar for a DEMO based low-code platform rules processing engine, in: e. a. Cristine Griffo (Ed.), Advances in Enterprise Engineering XVI - 12th Enterprise Engineering Working Conference, EEWC 2022, Leusden, The Netherlands, November 2-3, 2022, volume 473 of *LNBIP*, Springer, 2022, pp. 33–52. URL: https://doi.org/10.1007/978-3-031-34175-5_3. doi:10.1007/978-3-031-34175-5_3.