A New Action Meta-model and Grammar for a DEMO Based Low-Code Platform Rules Processing Engine

David Aveiro[1,2,3](✉) and Vítor Freitas[1,3](✉)

[1] Technology and Innovation, ARDITI - Regional Agency for the Development of Research, 9020-105 Funchal, Portugal
`daveiro@uma.pt, vitor.freitas@arditi.pt`

[2] NOVA-LINCS, Universidade NOVA de Lisboa, Campus da Caparica, 2829-516 Caparica, Portugal

[3] Faculty of Exact Sciences and Engineering, University of Madeira, Caminho da Penteada, 9020-105 Funchal, Portugal

Abstract. We consider current Design and Engineering Methodology for Organizations (DEMO) Action Rules Specification to be unnecessarily complex and ambiguous. Even while using a "structured English" syntax similar to the one used in Semantics of Business Vocabulary and Business Rules (SBVR), such specifications are: incomplete while not containing enough ontological information to derive a functional implementation; and complex by containing mostly unneeded specifications. We propose a new meta-model for DEMO's Action Model in the form of an Extended Backus–Naur Form (EBNF) syntax which is being implemented in a prototype that directly executes DEMO models as an Information and Workflow System. This prototype includes an action engine that runs DEMO transactions and the enclosed actions specified in our approach. We are currently integrating Blockly in our solution to allow syntactically correct visual programming of our proposed new Action Rule language that includes constructs to evaluate logical conditions, update the state of internal or external information systems, obtain input and provide output (formatted with a 'What You See Is What You Get' (WYSIWYG) template editor) to users, among others.

Keywords: enterprise engineering · DEMO · meta model · action model · action rules

1 Introduction

Numerous studies find that many software projects fall short of end customers' initial expectations. From [1], where certain case studies were conducted, a survey of 800 IT managers [2, 3] revealed that 63% of software development projects failed, 49% went over budget, 47% cost more to maintain than anticipated, and 41% fell short of meeting user and business requirements.

C. Griffo et al. (Eds.): EEWC 2022, LNBIP 473, pp. 33–52, 2023.
https://doi.org/10.1007/978-3-031-34175-5_3

Dalal et al. examined a number of project failure-related reports that have been published and built a list of failure factors that are responsible for this high failure rate [4]. Unrealistic project objectives, incomplete requirements, a lack of stakeholder and user involvement, issues with project management and control, an inadequate budget, changing requirements, inconsistent requirements and specifications, a lack of planning, poor communication, and the use of new technologies for which software developers lacked the necessary experience and expertise are common causes.

An enterprise engineering method called DEMO [5] is linked to a strong body of theories which intend to address the challenges highlighted above. Despite how sound DEMO is in theory, there are still many legitimate concerns regarding its utilization. DEMO's Action Model (AM), which is hardly ever employed in projects, is one of the fundamental components and one of the theoretical foundations that is frequently overlooked in current practice [6]. This occurs despite the fact that the methodology's creator himself regards the AM as the most significant model and where all model information is contained in detail [5, 7]. It is regarded as the organization's differentiator model, or what makes it special. And from this model one can elicit all other three aspect models of DEMO.

In this paper we propose a new Action Meta-Model and Grammar for a DEMO based low-code platform rules processing engine by evolving the DEMO Action Model with the proposal of a new meta-model in the form of a EBNF syntax which is currently being implemented in our DEMO based low-code platform, DISME (Direct Information Systems Modeller and Executer).

We claim that the way Action Rules are currently specified in DEMO, result in incomplete specifications that maintain ambiguity and do not contain enough ontological information for direct generation of information systems, as claimed by DEMO's propounder. With our proposal, we can describe, still on an ontological level, a wider range of crucial details and information, enabling a nearly direct execution of models as an information system. As a result, we help close the enormous gap between DEMO models and the significant implementation issues that surface during the software development process and which should be described right away along with ontological elements. Applying our proposal in a low code platform we are developing, by executing models directly, we drastically shorten the time it takes to produce information systems. And thanks to the use of DEMO as our core conceptual foundation, we have, as a starting point, a more complete elicitation of requirements, one of the main reasons Informations Systems projects fail. We demonstrate and validate our contribution using the EU-rent case [8].

2 Research Method

According to Design Science Research by A. R. Hevner [9, 10], the Information Systems Research paradigm used in this study should be viewed as a collection of three closely related cycles of activities.

On Fig. 1, these activities are depicted. Hevner argues that these three activities should not be used separately because only together do they provide a solid design science research and can produce a reliable result. Our research, with regards to the

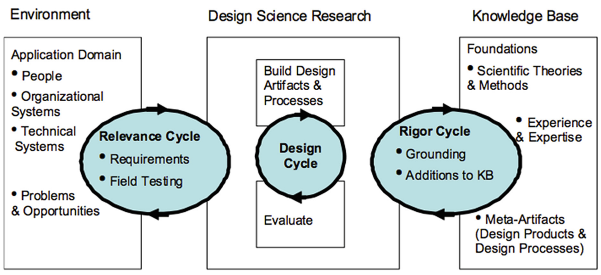


Fig. 1. Design science research cycles [10]

first cycle, Relevance, which is depicted in Fig. 1, revealed a glaring issue of ambiguity and a lack of concise and crucial information regarding the current syntax of DEMO Action Rules. As a result, an opportunity to design a more comprehensive syntax was at hand. We devised a new grammar for DEMO's Action Rules with relation to the second design cycle. This grammar was developed following numerous iterations of exhaustive and thorough design, implementation, and evaluation of various language elements, as well as testing them in the action executer engine in our prototype using both the EU-Rent case and a real-world project being developed in a nearby private company. We propose a new Action Meta Model for DEMO that, in our opinion, will allow the development of Action Rule Specifications in a more thorough and complete manner. Finally, the theoretical underpinnings of DEMO itself provide support for the studies about the final third cycle, Rigor.

3 Background and Theoretical Foundations

DISME uses DEMO methodology as a solid foundation for the production of collaborative-based organizational models and diagrams for the specification of its processes, information flow, responsibilities of both human and software, proce-dures and other kinds of organizational artifacts [11].

3.1 DEMO'S Operation, Transaction and Distinction Axioms

According to the operation axiom of the Ψ-theory [12], on which DEMO is founded, subjects in organizations execute two different types of acts: production acts that have an impact on the P-world, or production world, and coordination acts that have an impact on the C-world, or coordination world. Subjects are actors performing an actor role responsible for the execution of these acts.These worlds are always in a particular state indicated by the C-facts and P-facts that have transpired up to that point in time.

When active, actors consider the status of the P-world and the C-world. Actors continually strive to fulfill the agenda provided by C-facts. In other words, actors engage in interaction through the creation and management of C-facts. Figure 2 depicts this connection between the actors and the worlds. It illustrates the guiding principle of organizations whose members are dedicated to effectively accomplishing their agenda.

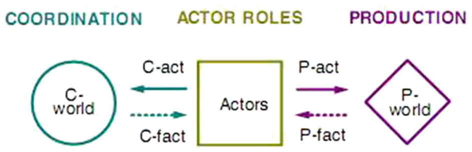


Fig. 2. Interaction of the Actor with the Production and Coordination Worlds [13]

The coordination actions are the means by which actors enter into and uphold commitments towards reaching a given production fact, whereas the production acts contribute to the organization's objectives by bringing about or delivering products and/or services to the organization's environment [14].

The coordinating acts follow a certain path along a generic universal pattern called transaction, in accordance with the transaction axiom of the Ψ-theory [12].

Three phases make up the transaction pattern: (1) the order phase, where the initiating actor role of the transaction expresses his wishes in the form of a request and the executing actor role promises to produce the desired result; (2) the execution phase, where the executing actor role actually produces the desired result; and (3) the result phase, where the executing actor role states the produced result and the initiating actor role accepts that result, effectively closing the transaction.

This succession, which is referred to as the "basic transaction pattern", only takes into account the "happy case", in which everything proceeds as predicted. To realize a new production fact, all five of these steps are essential. The universal transaction pattern that takes into account many more coordination acts, such as revocations and rejections that may occur at any point along the "happy path", is found in [14].

All transactions go through the four social commitment coordination acts of request, promise, state, and accept; however, these steps might be taken tacitly, that is, without any kind of explicit communication taking place. This could occur as a result of the adage "no news is good news" or just plain forgetfulness, both of which can seriously damage a business. Therefore, it's crucial to always take the complete transaction pattern into account while designing organizations. Two distinct actor roles are in charge of transaction steps. The request and accept phases are the responsibility of the initiating actor role, and the promise, execution, and state steps are the responsibility of the executing actor role. The responsible actor may not carry out these steps because the relevant subjects may delegate one or more of the transaction steps that fall under their purview to another subject, even if they are still ultimately liable for such acts [14].

3.2 DEMO Action Rules

DEMO Action Rules are the guidelines for managing events to which actors must react, or business rules. The Action Model of DEMO is not comprised by this set of rules alone, but also contains work instructions regarding the execution of production acts both represented in the Action Rules Specification (ARS) [7]. The Action Rule Specification (ARS) standard has evolved through time, starting with a pseudo-algorithmic language and culminating, in DEMO's specification language 4.5, in a definition which adheres to the Extended Backus-Naur Form (EBNF), the international standard syntactic meta language, defined in ISO/IEC 14977 [15].

The general form to represent an action rule is $<$ event part $>$ $<$ assess part $>$ $<$ response part $>$. What event (or collection of concurrent events) is reacted to is specified by the event part. An action rule's assess portion is divided into three sections that correspond to the three validity claims: the claims to rightness, sincerity, and truth. The final section, the response, is broken down into an if clause that outlines what must be done if the actor believes that complying with the event is justifiable and, potentially, what must be done if it is not. This method of developing action rules enables the performer to stray from the "rule" if they believe it is acceptable while also being held accountable for it [7].

We consider this way of Action Rules Specification to be ambiguous because, despite using a structured English syntax akin to that found in Semantics of Business Vocabulary and Rules [8], it does so in an imprecise manner that lacks some necessary ontological details to be used as the basis for the implementation of an information system. For instance, as we will discuss in more depth in Sect. 4, it lacks a method to deal with sets of actions or operators. Additionally, the current standard brings unneeded complexity since it includes a lot of extraneous details about three different forms of evaluation: fairness, sincerity, and truth. The following section, in which we go into more detail about our proposal will develop these claims.

4 Direct Information Systems Modeller and Executer

Three main components primarily make up DISME: 1) a Diagram Editor to create the higher level DEMO models in a graphical way 2) the System Manager to precisely detail and parametrize all DEMO Models, with a special attention to the Action Model, so that a complete information system can be specified according to an organization's demands; and 3) The System Executer to directly run the modeled information system in production mode.

In the System Manager, one or more users assume the administrator role and have the ability to modify each organizational process by creating and editing transactions, their relations, action rules and input forms that are associated with these transactions, in specific transactions steps, as well as by specifying entity and property types, that is, the main business objects and their attributes, or, in other words, the database of the information system. Users who model the system just need a basic understanding of enterprise engineering modeling, which is similar to the "language / representation" used within businesses, rather than requiring specific programming skills.

Users who have been granted authorization to participate in transactions in the System Executor do so in accordance with their roles and following DEMO's transaction pattern. The System Executor can be broken down into two main components: 1) the Dashboard, which serves as the user interface for users to interact with when performing organizational tasks, and 2) the Execution Engine, which controls the information and process flow in accordance with the full specification of the system.

The Dashboard interface can be seen in the following figures. In Fig. 3, it is shown where the user can start new processes, depending on the process types existing in the system and the current user's permissions. Here, it is also possible to see a section responsible for counting the pending and performed tasks, as well as delegations made.

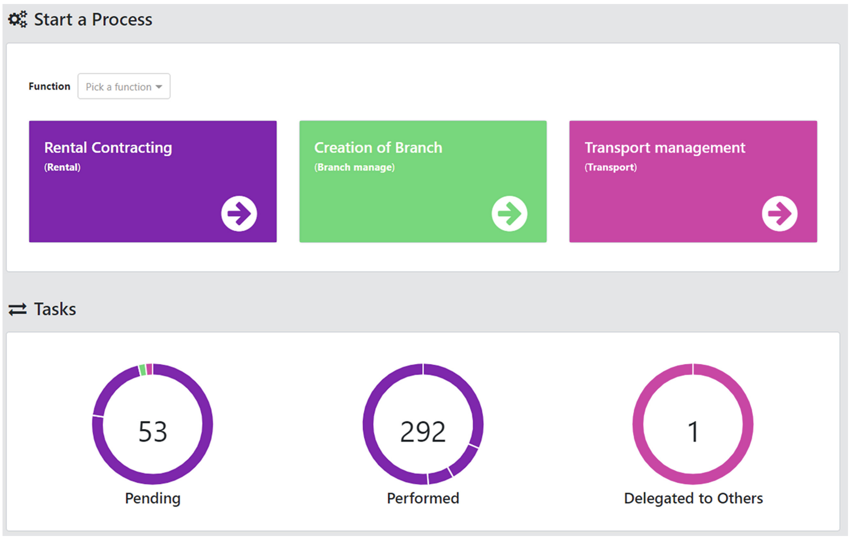
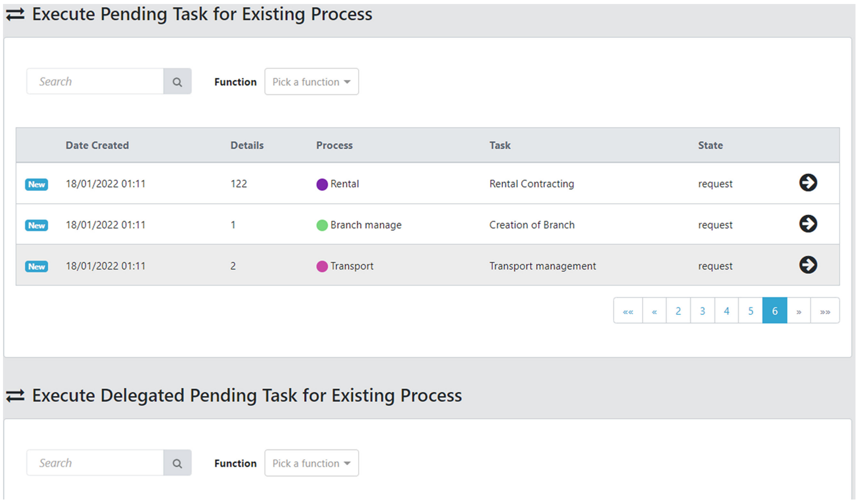


Fig. 3. Dashboard Interface - Start Process and Task Counting


Execute Pending Task for Existing Process
Search
Function
Pick a function
«« « 2 3 4 5 6 » »»
Execute Delegated Pending Task for Existing Process
Search
Function
Pick a function

	Date Created	Details	Process	Task	State
New	18/01/2022 01:11	122	Rental	Rental Contracting	request
New	18/01/2022 01:11	1	Branch manage	Creation of Branch	request
New	18/01/2022 01:11	2	Transport	Transport management	request

Fig. 4. Dashboard Interface - Pending Tasks

Figure 4 represents the Dashboard sections where users can look at their pending organizational tasks' data, such as creation date, the process it belongs to, the associated transaction type and state. Also, this is where the Execution Engine is incorporated so that the user can execute the tasks shown in the table rows.

The development of the database behind the prototype solution was heavily influenced by the DEMO way of thinking, trying to capture the essence of an organization's workflow, but without abstracting from their infological and datalogical implementations. One of the goals was to keep the platform as flexible as possible in terms of the editing possibilities available [16].

5 New Action Rule Syntax Specification and Implementation

In Table 1, we present, in EBNF[1], the current result of our iterations of development of a syntax and constructs specification of DEMO Action Rules which are runnable, in relation to its previous version [11]. We next introduce how this grammar corresponds to a set of requirements for the respective implementation of the DISME's engine that runs the action rules, and consequently, all the logic used for the implementation of their visual programming. In this specification presented in the table below, new concepts are highlighted in bold and updated ones are in italic.

An action rule occurs in the context of a transaction type, among those specified in the system, in the activation of a particular transaction state. An action rule can lead to the execution of one or more actions of a specific type. For example, an action may imply a causal link - changing the state of any transaction - or it may simply assign a value to a property in the system. We can have a sequence of one or more actions. For each action, one needs to specify the action type that will imply what concrete operations/instructions will be executed by the action engine and then define its parameters, specific to the corresponding action type, required for its execution.

An action can be specified that will prompt the user for input through a form, that is, for the user to *input* some data for a certain process instance. This form will be designed in the form management component of DISME, shown in Fig. 5, according to the *properties* associated with the respective action. It is also possible to specify, for each property in the form, *enabling conditions, validation conditions and form computing*. *Enable conditions* are used when we want that a property is "hidden/disabled" from the form unless the specified condition is true, which in that case the property will be shown. *Validation conditions* have to be satisfied/validated so that the user can submit the form data, being that if the condition is not satisfied, a message is presented back to him. *Form computing* enables us to define computations regarding data in the current form for a specific field, with that property being filled automatically based on the given expression instead of a manual fill by the user.

As opposed to the last action type mentioned, one can also define actions that will *output* information to the user. Using a WYSIWYG editor to create a new template or selecting an already saved template from the system's database, we can output a custom notification or dialog box directly to the user when the action rule is run. The possibility

[1] https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form

Table 1. Action Model EBNF specification (column separation equals the EBNF symbol " = ")

when	WHEN transaction_type IS|**HAS-BEEN** *transaction_state* { action} -
transaction_type	STRING
transaction_state	REQUESTED | PROMISED | EXECUTED | DECLARED | ACCEPTED | DECLINED | REJECTED | REVOKE_REQUEST_REQUESTED (…[2])
action	*causal_link* | assign_expression | *user_input* | **edit_entity_instance** | *user_output* | produce_doc | if | *API_CALL*
user_output	STRING
produce_doc	static_template | form_template
static_template	STRING
form_template	STRING
assign_expression	property " =" (term | property_value)
property	STRING
causal_link	transaction_type MUST BE transaction_state [min [max]] **[CANCEL_PROC] [CONTINUE_IF_SAME_USER]**
min	Integer
max	Integer | *
user_input	{ form_property}-
edit_entity_instance	{entity_detail} { form_property}-
form_property	property [form_calculation] [enable_condition] {validation_condition} [MANDATORY]
entity_detail	property
form_calculation	compute_expression
enable_condition	ENABLE condition
validation_condition	[NOT] validation_condition_type user_output
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}-

(*continued*)

[2] All other c-facts of the transaction pattern are here, but omitted for space reasons.

Table 1. (*continued*)

when	WHEN transaction_type IS|**HAS-BEEN** *transaction_state* { action} -
compute_operator	“ +” | “ -” | “*” | “/” | “^”

if	IF condition
	THEN { action} -
	[ELSE { action} -]

condition	(ISTRUE | NOT evaluated_expression | condition) | (AND | OR { evaluated_expression | condition}-)
evaluated_expression	comp_evaluated_expression | user_evaluated_expression
comp_evaluated_expression	term logical_operator term | property_value
user_evaluated_expression	STRING
logical_operator	“ <“ | “ >“ | “ = =“ | “! =“
property_value	STRING
term	constant | value | property | query | compute_expression | produce_doc
constant	value_type STRING
value	value_type STRING
value_type	TEXT | INTEGER_NUMBER | REAL_NUMBER | BOOLEAN | ENUM | DATE | TIME
query	STRING { term}
while	WHILE condition { action} -
foreach	FOREACH set { action} -
set	“set of elements”

to add properties to this editor, whose value is filled in the running of the action rule, thus making this a dynamic template, isn’t yet implemented but is planned to be included in a future iteration of the DISME.

It is also possible to specify ‘*if then else*’ flows, and in the *condition* one can specify complex conditions containing *logical condition*s evaluated automatically by the executor engine or *informal expressions* evaluated by the human user responsible for the transaction step as true or false, or a combination of both. *While* and *For each* kinds of flows are not yet implemented in the prototype but are also planned to be included in a future iteration.

The terminal symbols presented as string and set of elements are automatically parsed and interpreted by the action engine of DISME. The *set of elements* can be a group/array of elements that can be obtained from a customized query that returns a set of elements from the internal and/or external information system.

An important innovation in the action rules syntax is the realization that one needs to decompose a “normal” action rule into two action rules for each transaction state,

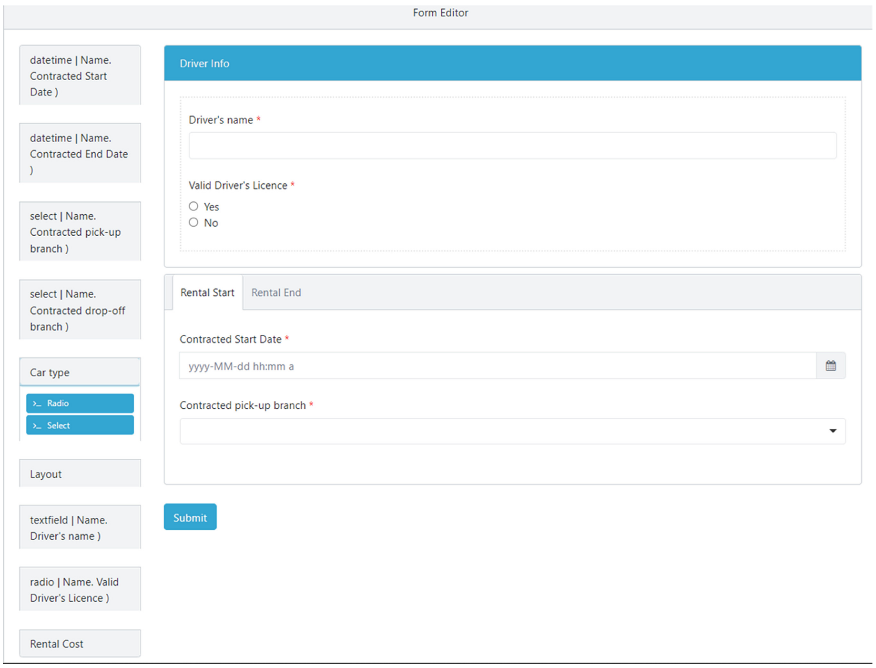

Form Editor
datetime | Name. Contracted Start Date)
datetime | Name. Contracted End Date)
select | Name. Contracted pick-up branch)
select | Name. Contracted drop-off branch)
Car type
Radio
Select
Layout
textfield | Name. Driver's name)
radio | Name. Valid Driver's Licence)
Rental Cost
Driver Info
Driver's name *
Valid Driver's Licence *
Yes
No
Rental Start
Rental End
Contracted Start Date *
yyyy-MM-dd hh:mm a
Contracted pick-up branch *
Submit

Fig. 5. Form Editor

one regarding the act itself and another for the respective fact created. This duality is achieved with the usage of the *IS* and *HAS-BEEN* terms when defining the root of the action rule, as can be seen in the first row of the EBNF table. We came to this realization while noticing that an actor, while executing a certain c-act or the p-act itself, will need to create some original fact(s) (e.g. input in a form while executing a request); and while dealing with a c-fact/p-fact having been executed, it might be needed to have complex conditions evaluation and also new facts creation or computation. It is also worth noting that the responsible role for the HAS_BEEN action rule is the opposite from the one responsible for the IS action rule while in the same transaction state, that is, if it is the initiating role of the transaction that is responsible for the IS action rule, it will be the executing role that will be responsible for the HAS_BEEN action rule of the same transaction state, and vice-versa. This allows an even more clear separation of responsibilities in DEMO models.

Another relevant addition to this syntax is the inclusion of the new *edit entity instance* action type. By carrying out demonstrations and trials of DISME's usage on information systems in a real scenario, it became apparent that an action for editing previously filled data, more specifically entity instances, was needed, especially in a data intensive, and not so process intensive, information system. With this action, the user can specify Action Rules that comprise the modification of editable properties, that is, properties that have the 'editable' flag active, belonging to entity instances created formerly in the current process instance. Also, properties, or entity details, can be specified to be shown in the entity selection modal's select box that will appear on the execution of this action type, in order to give context and facilitate the selection. The transaction type 'Edit Car Information' is an illustration of this. It allows one to change properties like a car's rental pricing and mileage. When creating a transaction instance of this type, the vehicle being edited has to be chosen from a dropdown list. Here, the entity details are the chosen characteristics that would be listed underneath each option, such as its color, to further specify which Car it belongs to, allowing the user to choose the appropriate option, for instance, if there were two identical cars.

Some flags were also added to the *causal link* action specification that handle how the executor engine should behave when running this type of actions, namely the '*cancel process*' and the '*continue if same user*' flags, that refer to whether the causal_link action cancels the current process, for example on the passage of a transaction to the 'quit' state, and whether the execution engine should take the user directly to the execution of the transaction step specified in the causal link, when it reaches this action, in case the current user in the engine's thread is also responsible for that step. The latter flag could be applied, for example, to the first causal link depicted on Fig. 6. In this scenario, due to this causal link action, if the car was deemed to be damaged, the Execution Engine would automatically execute the action rule related to the task "Damage Handling is Requested" as soon as it was generated, instead of continuing the execution of this action rule with the evaluation of the next 'if' statement's condition. This is very important in terms of usability, since the process can flow naturally between different transactions without the user needing to go back to his main dashboard and search for the new action rule that needs his or her input. In this type of action, *minimum* and *maximum* are optional and by default come pre-filled as 1. They indicate how many transactions should result from the current action, whereas if minimum doesn't exist, by default is equalled to 1 and.if maximum doesn't exist, by default is equalled to minimum.

```
WHEN 'Car drop-off' HAS_BEEN stated
IF ['car is damaged']
THEN
  ASSIGN_EXPRESSION 'car damage' = true
  CAUSAL_LINK 'Damage Handling' [must be] requested
ELSE
  ASSIGN_EXPRESSION 'car damage' = false
IF ['current date' > 'contracted drop-off date']
THEN
  ASSIGN_EXPRESSION 'late return penalty' = true
  ASSIGN_EXPRESSION 'late return penalty charge' = EXPRESSION
ELSE
  ASSIGN_EXPRESSION 'late return penalty' = false
IF ['Actual drop off branch' == 'Contracted drop-off branch]
THEN
  ASSIGN_EXPRESSION 'location penalty' = false
ELSE
  ASSIGN_EXPRESSION 'location penalty' = true
  ASSIGN_EXPRESSION 'location penalty charge' = EXPRESSION
IF ['late return penalty' == true OR 'location penalty' == true]
THEN
  CAUSAL_LINK 'Penalty Payment' [must be] requested
```

Fig. 6. Action rule to handle the transaction step 'Car drop-off has been stated'.

An example of an action rule definition, adapted to the last iteration of our Action Rule's Syntax, can be seen in Fig. 6.

After the first 'if' statement, an informal expression that needs to be evaluated by a human user physically inspecting the car and comparing it to the damage sheet signed at pickup can be found. In the event that there is freshly observed damage on the vehicle, a boolean property in the rental instance gets the value true written to it before a transaction to handle the issue is requested. This property serves as a flag in the rental entity and can then later be used to make general queries about rentals with or without damage. We then have a couple 'if' actions that automatically evaluate whether penalties should be applied. In case penalties are to be applied, mathematical expressions can be specified to calculate them automatically, by the engine, that take into consideration properties from the current process. In determining whether there is a location penalty, one can see the usefulness of having our "dual" specification of action rules for each transaction stat. In this case we can be sure to have the organizational facts that originated from the 'Car drop-off IS stated' transaction step which are then needed in this HAS-BEEN action rule. More specifically the 'Actual drop off branch' property would be a fact produced in the IS action rule and not in the HAS-BEEN rule. This need of having actions and facts in both "parts" of a transaction state was "disguised" with the term "WITH" in the current and limited ARS of DEMO. To conclude this action rule, we have another 'if' statement that starts a transaction to handle the penalty payment if needed.

6 DISME'S Components

Three components were implemented in DISME to enable the implementation of our new action rule format: 1) Action Rules Management, 2) Templates management, 3) Forms Management. Then, an Execution Engine was developed to automatically run action rules defined in the former components and a Dashboard was created that integrates its functionality and provides the interface with which users interact in organizational tasks. Due to space limitations, focus will be given in this paper to the Action Rules Management Component and to the Execution Engine.

6.1 Action Rules Management Component

In order to define a component that allowed the visual programming of these Action Rules, the Blockly library was used. Blockly is a library that adds a visual code editor to web and mobile applications. The Blockly editor uses interlocking, graphical blocks to represent code concepts like variables, logical expressions, loops, and more. It allows users to apply programming principles without having to worry about syntax or the intimidation of a blinking cursor on the command line [17]. It thus allows, as is the goal of this component in DISME, managers or individuals in a comparable position in organizations to develop Action Rules that are then saved and used in the execution engine through the Dashboard component, even if they have little or no prior programming experience. The choice of this library was also due to the fact that it is compatible with all the main browsers, i.e. Chrome, Firefox, Safari, Opera, and IE and that it is highly customizable and extensible [18].

This component is responsible for the creation, editing and consequent storage of action rules for a transaction type in a specific transaction state. Another important feature available on this component is one that allows us to see all previously created action rules and, if needed, load them onto the visual programming editor for editing.

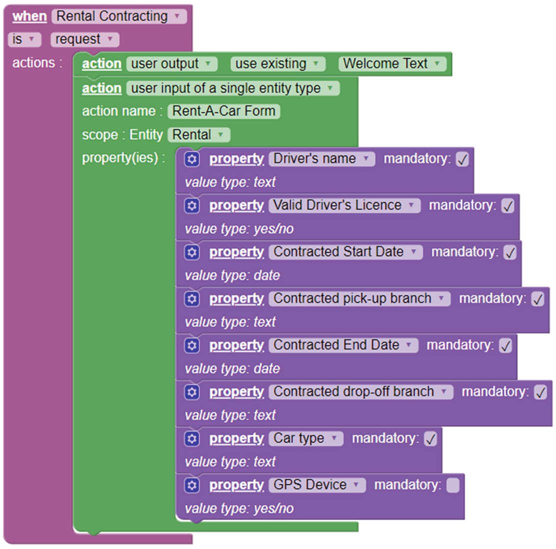

when Rental Contracting
is request
actions : action user output use existing Welcome Text
action user input of a single entity type
action name : Rent-A-Car Form
scope : Entity Rental
property(ies) : property Driver's name mandatory: ✓
value type: text
property Valid Driver's Licence mandatory: ✓
value type: yes/no
property Contracted Start Date mandatory: ✓
value type: date
property Contracted pick-up branch mandatory: ✓
value type: text
property Contracted End Date mandatory: ✓
value type: date
property Contracted drop-off branch mandatory: ✓
value type: text
property Car type mandatory: ✓
value type: text
property GPS Device mandatory: ☐
value type: yes/no

Fig. 7. Design of an action rule using the visual programming component.

An example of the definition of an action rule using this component can be seen in Fig. 7. This example represents the first transaction on a Rental process, that is, the presentation of a welcoming text to the user and then the filling of a form containing the main information from the rental and the renter.

6.2 Execution Engine

The Execution Engine has the function of executing action rules previously defined in the action rules management component through visual programming. This component had already been developed in a previous iteration of the DISME prototype, under the name of 'Expression Engine', due to the fact that its development was more focused on the evaluation of formal expressions. However, it was decided to restructure it due to the expansion of the requirements, previously expressed in EBNF syntax, and, consequently, of the system's database, which significantly affected this component and made its operationalization easier and more efficient.

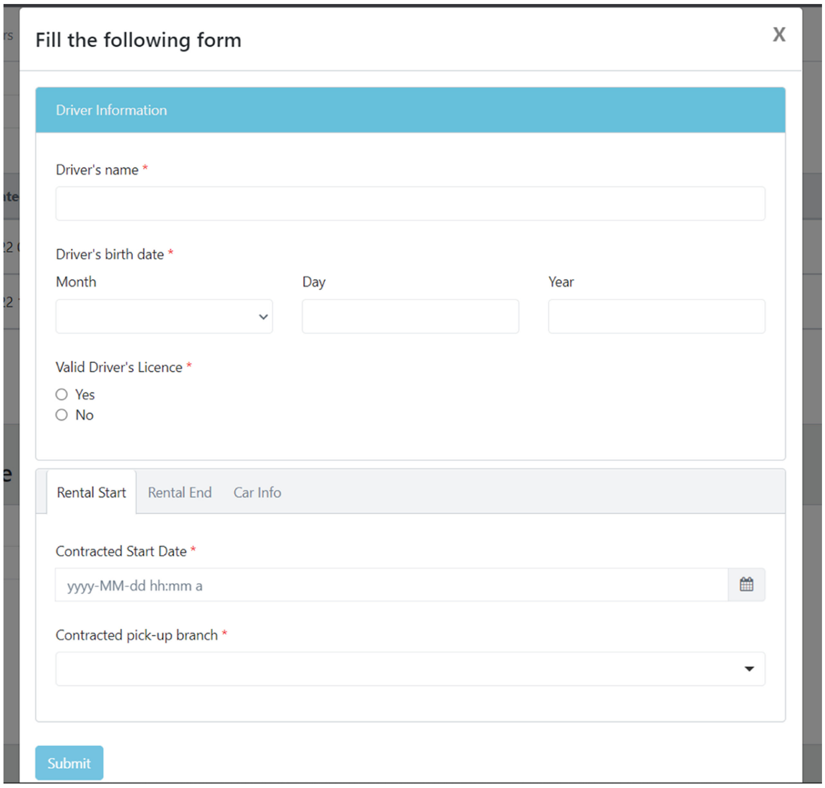
The Execution Engine is called when a user wants to execute an organizational task, i.e. a set of actions of a transaction in a specific transaction state, through the Dashboard component that provides the interface with which users interact for executing organizational tasks which they are responsible for. When called, the Execution Engine checks if this is a task whose execution is currently starting, in which case it fetches the first action of the action rule associated to it, or if it is a task whose execution has already started, in which case it fetches the action that was pending in the last execution or the action next to the last one performed, according to the action log that is generated by the DISME. Subsequently, it analyzes the type of action to be evaluated, and executes operations accordingly.

Also worth mentioning is that it distinguishes between two major action types: 1) Automatic Actions, which are executed automatically by the Execution Engine, and comprise actions like those of type ‘assign expression’, ‘causal link’ and ‘if - evaluation of logical conditions’; 2) Actions that necessarily require humN intervention, namely ‘user input’, ‘user output’, and ‘if - evaluation of informal expressions’. When the Execution Engine is interpreting an action rule, it will execute the corresponding actions automatically until it encounters an action that requires user intervention. When it finds one of these actions, execution flow is returned to the user for its intervention. After doing so, the automatic execution resumes until it finds another action needing human input or the action rule comes to its end.

We’ll now demonstrate and give an example of how the dashboard interface uses the execution engine to run an action rule definition and manage its flow. We’ll apply the action rule shown on Fig. 7 for this example.

When a user wishes to execute this ‘Rental Contracting IS requested’ organizational task, that corresponds to the first transaction to be run on a ‘Rental’ process, it will fetch the first action of this action rule. This corresponds to a ‘user output’ action, so the Execution Engine gets the template associated with the action, in this case a dialog box with a welcoming message, and displays it to the user. Then, when the user presses the dialog box’s button, completing this action, the Execution Engine will proceed to the execution of the next action of the action rule, which in this case is of the ‘user input’ type, that is, the presentation of a form to the user for filling in the properties (purple blocks) specified in Fig. 7. The structure of the form to be presented to the user is previously defined in the respective forms management component, after the specification of the action rule. An example of the result of the execution of an action of this type can be seen in Fig. 8 below.

When a user successfully submits the form, the execution engine detects that this is the final action listed in the action rule for that organizational task, marks it as finished, and, following DEMO’s standard flow of a transaction, automatically follows a default causal link which starts a new task corresponding to the next transaction state - “HAS-BEEN requested”. In case this new task requires human intervention, it will appear in the Dashboard of the users with the organizational role authorized to execute it. If not, the engine proceeds with the automatic execution of the action rules, following the DEMO flow, until it encounters an action that requires human intervention or the transaction comes to an end.


Fill the following form

Driver Information

Driver's name *

Driver's birth date *

Month	Day	Year

Valid Driver's Licence *

○ Yes
○ No

Contracted Start Date *

yyyy-MM-dd hh:mm a

Contracted pick-up branch *

Fig. 8. Execution of a ‘user input’ action in the Dashboard.

7 Discussion

The specification of Action Rules is created according to the following structure in the current official standard: $<$ event part $>$ $<$ assess part $>$ $<$ response part $>$. Although it is mentioned [7] that action rules established with the grammar of “structured English” are incredibly simple, it is also stated that some board members appeared perplexed when an action rule with this grammar was presented to them.

One of the grammar’s issues lies at the core of its specification. The formulation of these action rules appears to be excessively formal and challenging to comprehend for persons outside the scope of DEMO theory, as well as for new and inexperienced DEMO users.

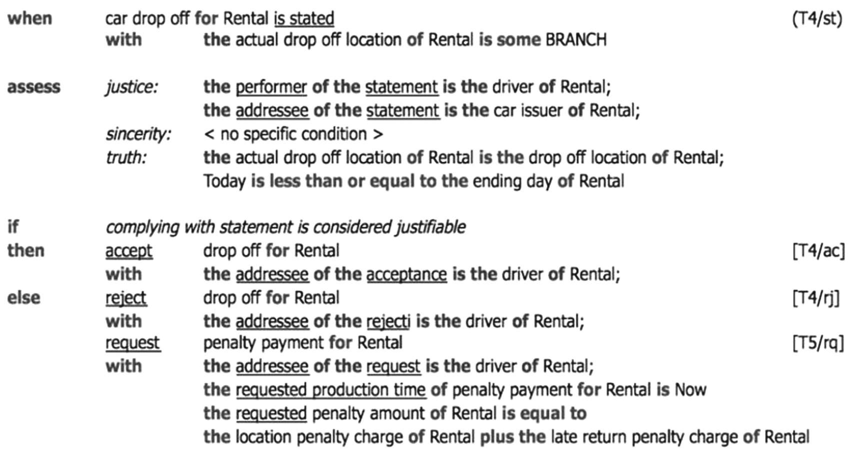
Comparing it to our approach, we may define a series of actions for an action rule, each with a particular type that indicates what the system should execute/perform in a simpler, literal, structured, and systematic manner, focused on implementation. We contend that the concepts of claims to rightness, sincerity, and truth mentioned in the $<$ assess part $>$ add unneeded ambiguity and complication. With our solution, one can specify a group of structured actions inside an action rule that have an immediate impact on the information system being developed by controlling the necessary process flows, and respective state changes and facts creation. This makes it easier and more effective for collaborators, such as system analysts, who are not aware of the social side of DEMO theory as articulated in the claims about rightness, truth, and sincerity, to comprehend and develop action rules. These claims make it harder to understand and develop action rules that can be fairly complex even with our grammar, as illustrated in Fig. 6.

Compared to the present standard, our grammar is more adaptable and includes a wider range of options and functionalities. For instance, we can specify inputs and outputs to the user, such as prompting a form or displaying information, common actions performed for an organizational process's successful functioning. Our proposal eliminates unnecessary details and complexities of official DEMO ARS, on the other hand it adds complex details which are nevertheless essential for implementation, but thorough visual specification of action rules which can be considered low-code. When pairing our language's straightforward constructions and visual programming component, collaborators such as analysts can specify/design action flows without the need for deep DEMO theory or technical programming knowledge, with DISME's execution engine then interpreting and executing them automatically, thus making their information system fully operational.

Ontology deals with the essence of reality and DEMO theories talk about 'implementation models' derived from higher level ontological models, but implementation models are also ontological. Our extensions of the DEMO meta-model with concepts such as documents, forms, value types, etc. are detailing essential aspects of implementation, but still agnostic of specific IT implementations (say, specific database, web server, client-side language, etc.). We have our DISME platform, but the models stored in its database could be perfectly run by another platform.

We will now go into greater depth about a few aspects of the two action rules' grammar. An action rule presented in the 'structured English grammar' format may be seen in Fig. 9. The $<$ assess part $>$ does not specify causal relationships in its numerous criteria. Due to the fact that we assess and check properties that correspond to a certain entity type connected to the current action being conducted in a straightforward manner, this does not occur in our grammar.

Regarding the $<$ truth $>$ claim, there is no way to specify the outcomes that may happen if each of the conditions is not met. Various actions may be executed in response to various circumstances, and various values may need to be updated, as seen in our example in Fig. 6. Figure 9 and Fig. 6 can be compared, and it is clear that syntax and simplicity are not the strong points of the current DEMO Action Rules' grammar. Additionally, nowhere in the action rule is it stated what consequences may occur if the "Actual drop-off branch" differs from the "Contracted pick-up branch". This action rule, defined in our grammar, as is described in Fig. 6, does not result in this uncertainty, as depending on whether certain conditions are true or untrue, we can describe multiple outcomes. In our case, we can call two different transactions in a way that is not allowed using current DEMO's syntax. Different action types can be specified in our grammar for an action rules' actions, but in this particular case, they are of type 'assign expression', as shown in Fig. 6. In this scenario, if we end up inside the ELSE block, the rental's "location penalty" property will automatically have its value set to "true" whereas the "location penalty charge" property will get its value from a mathematical expression,



```
when     car drop off for Rental is stated                                        (T4/st)
         with        the actual drop off location of Rental is some BRANCH

assess   justice:    the performer of the statement is the driver of Rental;
                     the addressee of the statement is the car issuer of Rental;
         sincerity:  < no specific condition >
         truth:      the actual drop off location of Rental is the drop off location of Rental;
                     Today is less than or equal to the ending day of Rental

if       complying with statement is considered justifiable
then     accept      drop off for Rental                                          [T4/ac]
         with        the addressee of the acceptance is the driver of Rental;
else     reject      drop off for Rental                                          [T4/rj]
         with        the addressee of the rejecti is the driver of Rental;
         request     penalty payment for Rental                                   [T5/rq]
         with        the addressee of the request is the driver of Rental;
                     the requested production time of penalty payment for Rental is Now
                     the requested penalty amount of Rental is equal to
                     the location penalty charge of Rental plus the late return penalty charge of Rental
```

Fig. 9. EU-Rent Action Rule TEOO [7]

which can be an operation between several values, two or more different properties, or a mix of the two.

In the < truth part > displayed in Fig. 9, when an action rule calls for other transactions it is not immediately clear which specific condition initiates the call to those transactions or how to manage information, inputs, and outputs. How to perform something of this sort in the TEOO [7] grammar is not at all clear. Many elements of the action rule are redundant or ambiguous, particularly those that begin with the 'with' clause or the rightness claim lines. The addressees and requested production time of a transaction, for example, should not need to be specified as they are already included in the context of the process instance that is carrying out these actions. These add unneeded complexity to the action rule. Figure 6 illustrates how our grammar makes it much simpler to understand what conditions and actions call for other transactions, such as the causal link "Penalty payment [must be] requested".

We also find that the use of the 'some' clause under the present standard brings ambiguity. In the case being examined, the context/instance should explicitly define the 'drop-off branch' at run time, negating the need for a distinct specification. DEMO models are purportedly designed to be independent of implementation and/or infological/datalogical considerations. In previous works we have been defending that DEMO models allow us to abstract from reality and reduce complexity, but they cannot be detached from reality/implementation, and action rules are the ideal place to recognize this relationship.

The DEMO Construction Model is quite detached from implementation since it provides a higher level and comprehensive view of a process as a tree of transactions and actor roles. But when it comes to business rules and execution, which are covered in DEMO's Action Rules, a more methodical and simple connection to reality/implementation is desperately needed. It is only natural that we "walk the last mile" and allow the specification of implementation details in action rules specification to the point of client output, database updates, and external calls to other systems, in a way that is independent of specific technology, as the current use of ‘with’ clauses is actually connecting to reality/implementation with clauses like ‘the requested production time of penalty payment is Now’ and also dealing with infological/datalogical issues with clauses like the one that defines the expression to calculate the penalty amount. So affirming that DEMO models should not include implementation aspects seems to be contradictory/illogical.

By carrying out demonstrations and trials of DISME’s usage on information systems in real projects scenarios, our grammar also greatly improved, with the main enhancement being the inclusion of a dual specification of action rules for each activated transaction state, with one regarding the act itself and another regarding the respective fact created, achieved with the usage of the IS and HAS-BEEN terms when defining the root of the action rule. With these demonstrations, it also became apparent that an action type for editing previously filled data, more specifically entity instances, was needed, especially when dealing with a data intensive, and not so process intensive, information system.

According to the GSDP mindset connected with DEMO theories [5], we are actually enabling a highly deep specification of the implementation model that, in a live system, like our DISME prototype, can be run immediately (without any compilation stages).

8 Conclusions and Future Work

As was mentioned above, the Action Rule Syntax we suggest in this paper is more thorough, flexible, and simpler to read, comprehend, implement, and run.

The Action Model is the ideal link between the implementation model and the higher level models (Construction Model and State Model), and is our DISME’s main focus. Our approach is superior because it explicitly states what actions will be executed, what inputs or outputs the system will produce, and what asynchronous calls to other transactions or information systems must be made.

The practical engineering approach we are using allows that, with minimal training on language constructs, specialized business analysts are able to scheme their organization’s flow in a way that effectively connects strategic high level models with low level details of implementation. These business analysts can then design action rules while also dealing with implementation issues like form design, user output, expression evaluation, and the information system’s flow control.

Our current prototype has some outstanding issues, such as allowing the implementation of for/while flows, while making sure that infinite cycles are not met and the incorporation of dynamic elements in templates. We also anticipate that the size and complexity of our grammar will continue to evolve and grow as it has been since its beginning. However, the philosophy that we adhere to and that was discussed in this paper continues to appear to be a promising approach.

In the conference, the presentation of this paper generated lively questions and discussion regarding the needs of improvement in DEMO’s Action Meta-model. Most of them were a replication of points raised by the reviewers and our clarifications generated consensus. We adapted the contents of the paper and the replies to the reviewers, taking into account the discussions. One very interesting point raised for discussion was the imperative vs. declarative nature of the Action Rules and the different alternatives of specifying complex branches of actions according to the evaluation of different interconnected (or not) logical conditions. It was presented to the authors the notion that it is possible to specify different action rules for the same C-fact to comply with different conditions, in order to avoid complex if-then-else trees. However, having important business logic dispersed in more than one action rule, seems, in our view, to bring unneeded complexity and possible combinatorial explosion [19] in case of need of changes.

Acknowledgments. This work was supported by the Regional Development European Fund (INTERREG MAC), project Dynamic eGov MAC2/5.11a/359.

References

1. Dalal, S., Chhillar, D.R.S.: Case studies of most common and severe types of software system failure. Int. J. Adv. Res. Comput. Sci. Softw. Eng. **7** (2012)
2. Shull, F., et al.: What we have learned about fighting defects. In: Proceedings Eighth IEEE Symposium on Software Metrics, pp. 249–258 (2002). https://doi.org/10.1109/METRIC.2002.1011343
3. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. **28**, 183–200 (2002). https://doi.org/10.1109/32.988498
4. Ibraigheeth, M., Fadzli, S.A.: Core Factors for Software Projects Success. JOIV Int. J. Inform. Vis. **3**, 69–74 (2019). https://doi.org/10.30630/joiv.3.1.217
5. Dietz, J., Mulder, H.: Enterprise Ontology: A Human-Centric Approach to Understanding the Essence of Organisation. (2020). https://doi.org/10.1007/978-3-030-38854-6
6. Dumay, M., Dietz, J., Mulder, H.: Evaluation of DEMO and the Language/Action Perspective after 10 years of experience, 29 (2005)
7. Perinforma, A.P.C.: The Essence of Organisation An Introduction to Enterprise Engineering. Sapio Enterprise Engineering. - References - Scientific Research Publishing, Presented at the (2015)
8. Bollen, P.: SBVR: A Fact-Oriented OMG Standard. In: Meersman, R., Tari, Z., Herrero, P. (eds.) OTM 2008. LNCS, vol. 5333, pp. 718–727. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88875-8_96
9. Hevner, A., R, A., March, S., T, S., Park, Park, J., Ram, Sudha: Design science in information systems Research. Manag. Inf. Syst. Q. **28**, 75 (2004)
10. Hevner, A.: A three cycle view of design science research. Scand. J. Inf. Syst. **19**, (2007)
11. Andrade, M., Aveiro, D., Pinto, D.: Bridging Ontology and Implementation with a New DEMO Action Meta-model and Engine. In: Aveiro, D., Guizzardi, G., Borbinha, J. (eds.) EEWC 2019. LNBIP, vol. 374, pp. 66–82. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-37933-9_5
12. Dietz, J.L.G., Mulder, H.B.F.: The PSI Theory: Understanding the Operation of Organisations. In: Dietz, J.L.G. and Mulder, H.B.F. (eds.) Enterprise Ontology: A Human-Centric Approach to Understanding the Essence of Organisation, pp. 119–157. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-38854-6_8
13. Dietz, J.: Enterprise Ontology: Theory and Methodology. Springer, Berlin Heidelberg (2006)
14. Dietz, J.L.G.: On the Nature of Business Rules. In: Dietz, J.L.G., Albani, A., Barjis, J. (eds.) CIAO!/EOMAS -2008. LNBIP, vol. 10, pp. 1–15. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68644-6_1

15. 14:00–17:00: ISO/IEC 14977:1996, https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/02/61/26153.html, Last Accessed 2 Oct 2022
16. Andrade, M., Aveiro, D., Pinto, D.: DEMO based Dynamic Information System Modeller and Executer: In: Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management. pp. 383–390. SCITEPRESS - Science and Technology Publications, Seville, Spain (2018). https://doi.org/10.5220/0007230003830390
17. Introduction to Blockly | Google Developers, https://developers.google.com/blockly/guides/overview?hl=pt, Last Accessed 1 Jan 2022
18. Blockly | Google Developers, https://developers.google.com/blockly, last accessed 2022/10/01
19. Brocade Desktop: irua, https://repository.uantwerpen.be/desktop/irua, Last Accessed 19 Dec 2022