Bridging Ontology and Implementation with a New DEMO Action Meta-model and Engine

Magno Andrade[1(✉)], David Aveiro[1,2(✉)], and Duarte Pinto[1(✉)]

[1] Madeira Interactive Technologies Institute, Caminho da Penteada, 9020-105 Funchal, Portugal
magnoandrade43@gmail.com, duartenfpinto@gmail.com
[2] Faculty of Exact Sciences and Engineering, University of Madeira, Caminho da Penteada, 9020-105 Funchal, Portugal
daveiro@uma.pt

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 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 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 WYSIWYG template editor) to users, among others.

Keywords: Enterprise engineering · DEMO · Meta model · Action model · Action rules · Syntax · Workflow · Information systems · Requirements

1 Introduction

Many studies claim that information technology projects fail to meet initial expectations of end users. From [1], where some case studies were developed, a survey with 800 IT managers [2, 3], found that 63% of software development projects failed, 49% suffered budget overruns, 47% had higher than expected maintenance costs and 41% failed to deliver the expected business value and user's expectations.

In [4] from 2019, authors analyse many of the published papers regarding project failure and compile a list of failure factors contributing for this high failure rate. Some of the common causes were unrealistic project objectives, incomplete requirements, lack of stakeholder's and users engagement/involvement, problems in project management and control, insufficient budget, unrealistic expectations, changing

D. Aveiro et al. (Eds.): EEWC 2019, LNBIP 374, pp. 66–82, 2020.
https://doi.org/10.1007/978-3-030-37933-9_5

requirements, requirements and specifications inconsistency, lack of planning, lack of communication, use of new technologies that software developers didn't have adequate experience and expertise, amongst others.

DEMO [5] is a well-established enterprise engineering method associated with a solid collection of theories that aim to contribute to solving the before mentioned problems. However, regardless of how sound DEMO is in theory many open ends remain. One of the clearer examples is the models that are produced and used for isolated efforts for analysing the organization and providing support for discussing changes initiatives. Current practice very commonly leaves out one of the key pillars in the theory and one of the main components – the Action Model (AM) – which indeed is barely used in practice [6]. This happens even though the founder of the methodology himself has considered, in [5] and [7], the AM as the most important model and where all essential model information can be found. It is considered to be the differentiator model of the organization – what makes it unique – and, alone, can be used to derive the remaining three aspect models.

This paper is integrated in a broader research initiative that aims at the development of a software platform having the 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, procedures and other kind of organizational artefacts.

Those models and diagrams should provide an up to date "picture" of the "organizational self" at any given time and in a collaborative fashion, guiding its participants in, (1) supporting the perception of the global reality of the organization [8], (2) supporting the definition and execution of their operational work and (3) supporting the creative process for organizational change [9].

Like our initiative, other widely used approaches such as ArchiMate [10] and BPMN [11] try to tackle most of these goals but suffer from the lack of a solid formal theory behind them and from ambiguous semantics [12, 13].

Our DEMO-based approach, based on sound theory drew some inspiration from the Universal Enterprise Adaptive Object Model (UEAOM) [14], and aims at the generation and execution of DEMO models that capture crucial information of organizational responsibilities and the flows of information, often overlooked in other approaches. Using these easy to share and understand models, with a high level of abstraction, we systematically seek to derive increasingly detailed models for executable workflows and manual work instructions.

In this paper we propose Bridging Ontology and Implementation with a new DEMO Action Meta-Model and Engine by revising the DEMO Action Model and proposing a new meta-model in the form of a EBNF syntax which is currently being implemented in our prototype called DISME (Direct Information Systems Modeller and Executer).

We claim that the current way of specifying Action Rules in DEMO leads to incomplete specifications that, on one hand, do not contain enough ontological information and, on another hand, keep a reasonable amount of ambiguity. With our proposal, we can specify – still on an ontological level – a broader variety of essential details and information to allow an almost direct execution of models. Thus, we contribute to bridge the huge gap between DEMO models and the important

implementation problems that arise at project time and which must be specified immediately in conjunction with ontological elements. By combining our approach with a low code platform prototype that we are developing, we aim to contribute to bridge and solve the gaps mentioned in the first paragraphs of this introduction. By having a direct execution of models we highly reduce the time to production of information systems and by using DEMO as a base we have as a starting point a more complete elicitation of requirements, one of the main points of failure in IS projects. We use the EU-rent case presented in [15] to exemplify and validate our contribution.

2 Research Method

The Information Systems Research paradigm adopted in this paper should be considered as a group of three closely related cycles of activities according to Design Science Research by Hevner [16, 17].

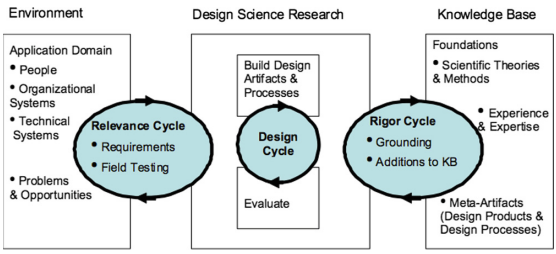

Fig. 1. Design science research cycles [17]

These activities are represented on Fig. 1. Hevner claims that only together these three activities constitute a good design science research and could render a valid output and therefore should not be applied isolated. In our research, and in relation to the first cycle, Relevance, represented in Fig. 1, we identified a clear problem of ambiguity and lack of concise and essential information about the current syntax of the DEMO Action Rules and therefore an opportunity to design a more comprehensive syntax was at hand. In regards of the second cycle of design, we propose a new grammar for DEMO's Action Rules. These rules were obtained after several iterations of exhaustive and comprehensive, design, implementation and evaluation of different grammar and language elements, as well as testing in the action executer engine in our prototype, both with the EU-Rent case and a practical project being developed in a local private company. We propose a new Action Meta Model for DEMO that we claim to allow a more concise, comprehensive and complete way for devising Action Rule Specifications. Finally, concerning the last third cycle, Rigor, the research is supported by the theoretical grounding foundations of DEMO itself.

3 Background and Theoretical Foundations

3.1 DEMO's Operation, Transaction and Distinction Axioms

In the Ψ-theory [18] – on which DEMO is based – the operation axiom [5] states that, in organizations, subjects perform two kinds of acts: production acts that have an effect in the production world or P-world and coordination acts that have an effect on the coordination world or C-world. Subjects are actors performing an actor role responsible for the execution of these acts. At any moment, these worlds are in a particular state specified by the C-facts and P-facts respectively occurred until that moment in time. When active, actors take the current state of the P-world and the C-world into account. C-facts serve as agenda for actors, which they constantly try to deal with. In other words, actors interact by means of creating and dealing with C-facts. This interaction between the actors and the worlds is illustrated in Fig. 3. It depicts the operational principle of organizations where actors are committed to deal adequately with their agenda. The production acts contribute towards the organization's objectives by bringing about or delivering products and/or services to the organization's environment and coordination acts are the way actors enter into and comply with commitments towards achieving a certain production fact [19].

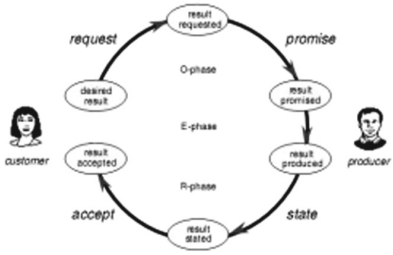

Fig. 2. Basic transaction pattern [5]

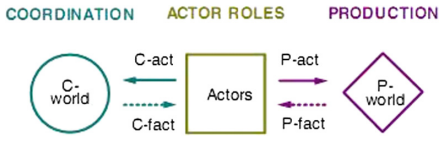

Fig. 3. Actor's Interaction with production and coordination worlds [5]

According to the Ψ-theory's transaction axiom, the coordination acts follow a certain path along a generic universal pattern called transaction [5].

The transaction pattern has three phases: (1) the order phase, were the initiating actor role of the transaction expresses his wishes in the shape of a request, and the

executing actor role promises to produce the desired result; (2) the execution phase where the executing actor role produces in fact 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, thus effectively concluding the transaction.

This sequence is known as the basic transaction pattern, illustrated in Fig. 2, and only considers the "happy case" where everything happens according to the expected outcomes. All these five mandatory steps must happen so that a new production fact is realized. In [19] we find the universal transaction pattern that also considers many other coordination acts, including cancellations and rejections that may happen at every step of the "happy path".

Even though all transactions go through the four – social commitment – coordination acts of request, promise, state and accept, these may be performed tacitly, i.e. without any kind of explicit communication happening. This may happen due to the traditional "no news is good news" rule or pure forgetfulness, which can lead to severe business breakdown. Thus the importance of always considering the full transaction pattern when designing organizations. Transaction steps are the responsibility of two specific actor roles. The initiating actor role is responsible for the request and accept steps and the executing actor role is responsible for the promise, execution and state steps. These steps may not be performed by the responsible actor as the respective subjects, may delegate on another subject one or more of the transaction steps under their responsibility, although they remain ultimately responsible for such actions [19].

The distinction axiom from the Ψ-theory states that three human abilities play a significant role in an organization's operation: (1) the *forma* ability that concerns datalogical actions; (2) the *informa* that concerns infological actions; and (3) the performa that concerns ontological actions [5]. Regarding coordination acts, the performa ability may be considered the essential human ability for doing any kind of business as it concerns being able to engage into commitments either as a performer or as an addressee of a coordination act [19]. When it comes to production, the performa ability concerns the business actors. Those are the actors who perform production acts like deciding or judging or producing new and original (non derivable) things, thus realizing the organization's production facts. The informa ability on the other hand concerns the intellectual actors, the ones who perform infological acts like deriving or computing already existing facts. Finally, the forma ability concerns the datalogical actors, the ones who perform datalogical acts like gathering, distributing or storing documents and or data. The organization theorem states that actors in each of these abilities form three kinds of systems whereas the D-organization supports the I-organization with datalogical services and the I-organization supports the B-organization (from Business=Ontological) with informational services [20].

3.2 DISME (Direct Information Systems Modeller and Executer)

DISME [21] is mainly comprised by three modules: (1) a Diagram Editor to create and view DEMO models as well as import and export them to the System Modeller (2) the

System Modeller to adapt and parametrize in detail the information system to the needs of the organization; and (3) the System Executer that runs in production mode the modelled information system.

In the System Modeller, one or more users take upon their selves the administrator role, and are able to shape each process of an organization creating and editing transactions, their relations as well as associating input forms to these transactions, or in specific transactions steps, together with the specification of entity and property types, that is, the main business objects and their attributes or, in other words the database of the system. Forms are dynamically generated by the System Executer component taking in account all specifications and when users are fulfilling their organizational tasks. The users that model the system, have no need for any specific programming skill only some basic knowledge of enterprise engineering modelling which is close to the "language /representation" used within organizations.

In the System Executer, users that have acquired permissions to take part in the transactions do so according to their roles following DEMO's transaction pattern.

The System Executer can be divided itself into two main components, (1) the Dashboard that provides the user interface with which the users interact with on their organizational tasks and (2) the Action Engine, that controls the flow of information according with the transactions and causal links defined in the Action Rules and Transactions and their relations (the equivalent to the Process Step Diagram).

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.

3.3 DEMO Action Rules

DEMO Action Rules are the specifications for handling events that actors have to respond to, 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 general form to represent an action rule is <event part> <assess part> <response part>. The event part specifies what event (or set of concurrent events) is responded to. The assess part in an action rule is divided in three sections, corresponding with the three validity claims: the claim to justice, the claim to sincerity, and the claim to truth. And the final part, the response, is divided in an if clause that specifies what action has to be taken if the actor considers complying with the event to be justifiable, and possibly what action must be taken if this is not the case. This way of formulating action rules allows the performer to deviate from the 'rule', if he/she thinks this is justifiable (and for which he/she will be held accountable) [7].

We consider this set of Action Rules Specification to be ambiguous because, although it uses a structured English syntax similar to the one used in Semantics of

Business Vocabulary and Rules [15] it does so in an incomplete way that does not contain all the needed ontological information to derive the implementation from it. For example, it lacks a way do deal with sets of actions or operators as we will approach in further detail on Sect. 3 New Action Rule Syntax – Specification and Implementation. And this set of rules is also complex by containing mostly unneeded specifications of three types of assessment, the justice, sincerity and truth. These claims are developed on the next section where we elaborate on our proposal. This is why we propose a new set of rules Bridging Ontology and Implementation with a new DEMO Action Meta-Model and Engine.

4 New Action Rule Syntax

In the following tables, we present, in EBNF, the current result of our iterations of development of a syntax and constructs for a runnable specification of DEMO Action Rules (Tables 1 and 2).

Table 1. EBNF specification for Action Model (the column separation means the "=") (Part 1)

when	"WHEN" transaction_type "IS" (c-fact | p-fact) action
transaction_type	string (NOTE: has to be a transaction specified in the system)
c-fact	"requested" | "promised" | "stated" | "accepted" | "revoke_request_requested" | "revoke_request_allowed" | "revoke_request_refused" | "revoke_promise_requested" | "revoke_promise_allowed" | "revoke_promise_refused" | "revoke_statement_requested" | "revoke_statement_allowed" | "revoke_statement_refused" | "revoke_acceptance_requested" | "revoke_acceptance_allowed" | "revoke_acceptance_refused" | "rejected" | "declined"
p-fact	"executed"
action	action_type [(assign_expression | causal_link)] {action }
action_type	specify_data | if | while | foreach | "C-ACT" | "WRITE_VALUE" | "READ_VALUE" | "PRODUCE_DOCUMENT" | "CLIENT_OUTPUT" | "EXTERNAL_CALL"
assign_expression	property "=" (constant_value | property_value | math_expression)
math_expression	string (NOTE: a mathematical expression evaluated by the dashboard, in principle produced by blockly mathblocks)
property	string (NOTE: has to be an existent property specified in the internal information system)
constant_value	String

Table 2. EBNF specification for Action Model (the column separation means the "=") (Part 2)

causal_link	transaction_type "[must be]" c-fact min max
min	Integer
max	Integer | *
specify_data	{property [cur_form_compute_code]} - {condition CLIENT_OUTPUT} (NOTE: for each pair (validation) condition+output if condition is true engine goes ahead if not shows CLIENT_OUTPUT)
cur_form_compute_code	ENABLE condition | math_expression
if	"IF" condition "THEN" action ["ELSE" action]
condition	["NOT"] evaluated_expression {("AND" | "OR") condition}
evaluated_expression	comp_evaluated_expression | user_evaluated_expression
comp_evaluated_expression	property operator (property_value | property)
user_evaluated_expression	String (NOTE: dashboard shows this "textual informal expression" that has to be evaluated by the user who will decide on a result of true or false)
property operator	"<" | ">" | "==" | "!=" | "∼"
property_value	string (NOTE: can be a numerical value or a possible ENUM value associated to a property)
while	"WHILE" condition action
foreach	"FOREACH" set action
set	set of elements

As we can see in the previous EBNF specification, an action rule occurs in the context of a transaction type 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. Namely, an action might imply a causal link or simply assigning some value to some 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. We can also express logical conditions that allow us to design expressions that are evaluated by the engine and determine the path that a certain process instance must take. We can specify an action that will automatically generate a form for user input, that is, for the use to specify some data for a certain process instance. This form will be automatically generated by the dashboard according to the properties associated to the respective action. It's possible to specify, for each property in the form a condition that has to be satisfied/validated so that the process can advance. If the condition is not satisfied it's possible to define a particular output to the user. It's also possible to define simple computations regarding data in the current form. One can also specify traditional "if then else" flows and logical conditions that are evaluated automatically by the engine to control the flow. It's possible also the formulation of "informal expressions" that have to be evaluated by the user as true or false

in order for the flow to continue in a certain way or another. While and for each kinds of flows are still not implemented but are planned to be included in our prototype.

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.

```
WHEN 'Car drop-off' IS stated
IF ['car is damaged']
THEN
      WRITE_VALUE 'car damage' = true
      C-ACT 'Damage handling' [must be] requested
ELSE
      WRITE_VALUE 'car damage' = false
IF ('current_date' == 'contracted drop-off date')
THEN
      WRITE_VALUE 'late return penalty' = false
ELSE
      WRITE_VALUE 'late return penalty' = true
      WRITE_VALUE 'late return penalty charge' = EXPRESSION
IF ('Actual drop-off branch' == 'Contracted drop-off branch')
THEN
      WRITE_VALUE 'location penalty' = false
ELSE
      WRITE_VALUE 'location penalty' = true
      WRITE_VALUE 'location penalty charge' = EXPRESSION
IF ('late return penalty' = true OR 'location penalty' == true
THEN C-ACT 'Penalty payment' [must be] requested
IF [car drop-off is considered justifiable]
THEN C-ACT 'Car drop-off' [must be] accepted
ELSE C-ACT 'car drop-off' [must be] rejected
```

Fig. 4. Action Rule to handle the state step of transaction Car drop-off

In Fig. 4 we can see an example of an action rule using our newly proposed syntax. After the IF we can find an expression that has to be evaluated by a human user looking physically at the car and comparing to the damage sheet signed at pickup. In case new damage is present a transaction is initiated to handle the issue, but before that, a property in the rental instance of boolean type has the value true written to it. This property works as a flag in the rental entity which is needed for general queries on rentals. We then have a couple of IF instructions where conditions can be automatically evaluated and enacted upon. Different other flags associated to the rental can be updated accordingly and the value of penalty charges can be calculated by mathematical expressions. The rule finishes with a couple more IF instructions, the first to determine if the penalty payment transaction must be requested and the final which provides another expression for the user to decide on the evaluation and eventually accept the transaction even if normally that would not be the case.

To enable the implementation of our new format for action rules in DISME, three components were implemented; (1) the Action Rules Manager, (2) the Action Manager, and (3) the Action Template manager. The above example applying our syntax can be specified in these components as to allow the engine to later interpret the rule. Action Rules Manager and Action manager – these are the components responsible for

creating, editing and deleting the action rules for each transaction type and transaction state. As mentioned above one action rule has one or more actions associated. On this component there are multiple functionalities available. We can create new actions associated to that action rule or view actions that were previously created. We next present in Fig. 5 a different and older version of the action rule above, equivalent to the version of the DISME's screen shot provided also ahead as the prototype is currently being adapted to the last version of our syntax presented in this paper.

```
WHEN 'Car drop-off' IS Stated
IF 'Contracted drop-off branch' != 'Actual drop-off branch' THEN
     ATOMIC c-act
          'Penalty payment' [must be] Requested
```

Fig. 5. Example of actions for the action rule handling the state of car drop-off

The above example works as follows, when the "Car drop-off" transaction is in the Stated transaction state, the action type IF is evaluated. If the condition (automatically evaluated by the engine) evaluates to true, the ATOMIC action type c-act is performed.

When using the component Action manager, actions that belong to the action rule selected by the user are displayed. Figure 6 shows the same action rule presented in Fig. 5, but according to the DISME interface. This is the main component where we can add different actions, logical conditions, templates, etc. While on execution mode of DISME, these actions will be interpreted by the engine developed together with the dashboard interface.

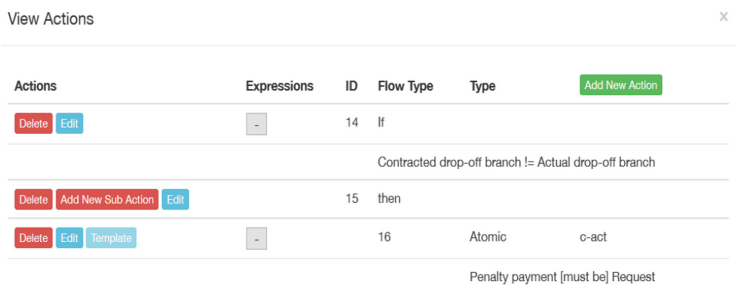

Fig. 6. List of actions that belongs to a specific action rule

Action Template Manager – this component is only used together with an action. The purpose of this functionality is to have custom templates for certain actions. Note that each action can have multiple templates but a template can only belong to an action. The engine in DISME uses the specifications inserted into these three components to apply, enforce and control the flow of the transaction and subsequently perform the specified actions. This means that when a user initiates or wants to continue a transaction inside the dashboard, if the transaction has actions associated, the

engine analyzes, verifies and interprets the actions associated to an action rule and carries out the respective operations in the context of that particular transaction instance. Some of these actions can be automatically performed by the engine without the need to wait for user input. Actions of type PRODUCE_DOCUMENT and CLIENT_OUTPUT use the templates specified in the Action Template Manager with a WYSIWYG editor that can use properties and values of the underlying information system to either automatically produce a PDF document or output formatted content to the client interface.

DISME allows the specification of entity types and properties in a database like fashion, allowing business users to, in a graphical user interface, specify their business objects and fields. In actions of type specify_data we can select a set of one or more existing properties that need to be specified by user input. The DISME dashboard will automatically render a form based on the input types defined previously in a form editor. After, for example, a successful reservation of a rental, the dashboard can output to the client some formatted text defined in a template and using elements of the filled form and/or other elements from the database.

We now illustrate and present an example of the dashboard interface using the action engine module that interprets our action rule specifications to control the flow.

In this example, we are using the action rule exhibited on Fig. 5, but first we need to explain the flow intended in this process. For this particular example, we will consider that we have only two transactions: T01 – Rental Contracting and T02 – Car drop-off. T02 has an action rule defined that is the same as on Fig. 6.

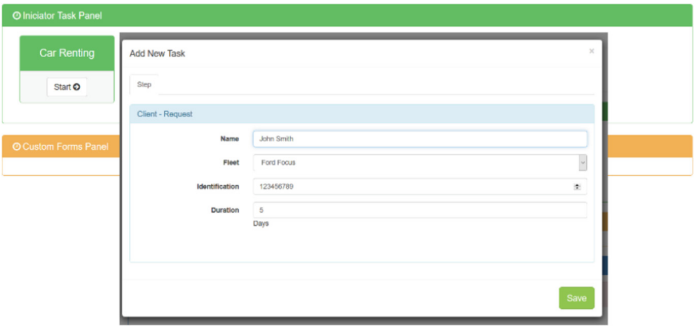

Fig. 7. Car Rental Request

In this process, the first transaction step to be performed is T01 – Request (Fig. 7). After that, transaction T02 is the next to be performed (state Request) and a field is filled by the user. When the state for the transaction T02 is "Stated" the engine module is going to retrieve previously inserted data that has been filled in T01 – Request. This data retrieved by the engine module is the data inserted previously on the field Contracted pick-up branch (T01 – Request) and evaluated with inputted data from another field. This field was filled by the user on the current transaction T02 but on another

state (T02 – Request). Thereafter, the engine module decides what path to proceed always having as a base the actions specified in the action rule.

Below we can see an example of the dashboard using these action rules:

Fig. 8. T01 - Rental Contracting and the property field Contracted drop-off branch

One user initiates the transaction T01 and selects the option Lisbon Airport for the field Contracted drop-off branch as shown in Fig. 8.

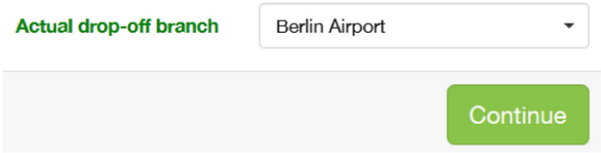

Fig. 9. T01 - Rental Contracting and the property field Contracted drop-off branch

After the initialization of T01, another user initiates the transaction T02 and selects the option Berlin Airport for the field Actual drop-off branch as show in Fig. 9 and presses the green button Continue. When the transaction state "Stated" for the transaction T02 is a fact, the engine will evaluate the expression that is inside rectangle 1 in the following Fig. 10:

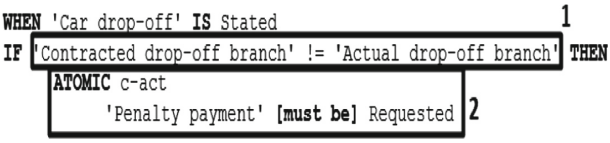

Fig. 10. Action Rule used by the engine module after the button Continue is pressed.

Given the example case, the expression will be evaluated as true because the chosen Contracted drop-off branch in T01 and Actual drop-off branch in T02 are different ('Lisbon Airport' ! = 'Berlin Airport') and so the path chosen by the engine module will be the instructions/operations/actions within the THEN block. The instructions within the THEN block are inside rectangle 2 in Fig. 10. With theses instructions/operations/actions, a new instance/transaction of transaction type "Penalty payment" on the state "Request" is created automatically by the engine.

5 Discussion

In the current official standard, the specification of Action Rules is devised with the following structure (see example in Fig. 11): <event part> <assess part> <response part>. Although in [7] it is claimed that action rules specified with the grammar of 'structured English' are very simple, it is also stated that when presenting this grammar to members of the board, some of them appeared to be confused with the explanation of this first action rule.

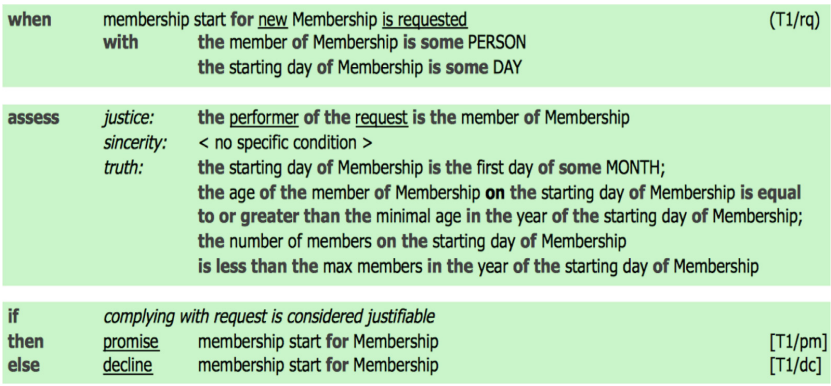

Fig. 11. Action Rule in 'structured English"

One of the problems of this grammar lies at the root of its specification. The author also states that the action rule should be written so we can understand them as formal, this arises that the formulation of these action rules is apparently too formal and difficult to read for people outside the scope of DEMO theory and even to new and inexperienced people using DEMO.

Comparing it to our approach, we can specify a set of actions for an action rule, each with a specific type which denotes what the system should execute/perform in a more simple, literal, structured and systematic way, already oriented to implementation. We argue that the notions of claims to justice, sincerity and truth specified in the <assess part> bring unnecessary complexity and ambiguity. With our approach, actions inside an action rule can be specified as a group of structured acts, some with direct effect on the information system with eventual associated expressions (logical and/or arithmetical) that control the flow of actions. This allows collaborators like system analysts, who are not aware of the social side of DEMO theory expressed in the truth, justice and sincerity claims, to understand and write action rules in a simpler and more powerful way. There is no need to complicate the action rules with the language action paradigm claims as, even with our structure, rules can become somewhat complex in some cases as the example we presented previously in Fig. 4.

Our grammar is more flexible and has many other options and functionalities as compared to the current standard. For example, we can perform inputs and outputs to

the client, such as producing documents or showing information which is necessary for the proper and informed functioning of the organization's process. Collaborators acting as analysts can specify actions with the simple constructs of our language which, in their essence and syntax, specify clearly what is intended with them and without the need of knowledge of technical programming languages. DISME's action engine automatically interprets the specified rules and we are currently adapting Google' Blockly platform to our prototype so it's even easier for business analysts to design action rules.

We will now discuss in a more specific and detailed way some aspects of the two grammars of action rules. On Fig. 12, another action rule in the format of the structured English grammar is displayed. The <assess part> doesn't specify causal links in the multiple conditions specified in this part, this does not happen in our grammar because we evaluate and verify in a simple way properties that belong to a certain entity type related to the current action being executed.

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. 12. EU-Rent Rule TEOO [7]

Regarding the <truth> claim there is no way of specifying the consequences that can occur if conditions present here are not individually fulfilled; different actions might need to be executed due to different conditions and different values might need to be updated like shown in our example in Fig. 4. If we compare Figs. 12 and 4, we can immediately conclude that syntax and simplicity are not the strength of the current grammar of Action Rules for DEMO and that it is not specified anywhere in the action rule what consequences can arise if the 'Actual drop-off branch' is not the same as the 'Contracted pick-up branch'. The same does not happen in the action rule defined in our grammar, as specified in Fig. 4. We can specify consequences for different conditions depending if they are either true or false. In our example we can call 3 different transactions in a way that would not be possible with current standard syntax.

As mentioned above, each action rule is a set of actions, so for each action inside the action rule, we also define what kind of action (action type) will happen at a certain

point in the action rule, for this particular case the consequences that can happen are of the action type WRITE_VALUE as shown on Fig. 4. In this case, if we are within the ELSE block, the 'location penalty' property of the rental will have its value assign automatically to 'true', and on the other hand, the 'location penalty charge' property will have its value obtained from an 'expression', for example this 'expression' can be a mathematical operation between two values, or even two distinct properties.

In the <response part>, that is displayed on the Fig. 13, when an action rule calls for other or multiple transactions, it is not immediately apparent not only which particular condition originates a call to other transactions nor how to handle information, inputs and outputs. It is not clear at all how to do something of this kind in the TEOO [7] grammar. Several parts of the action rule, especially the ones starting with the *with* clause or the justice claim lines are redundant or ambiguous. There should be no need to specify elements such as the addressees or requested production time of a transaction as those elements are automatically part of the context of an instance of a process performing these actions. These add unneeded complexity to the action rule.

```
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. 13. <response part> of the EU-Rent Rule of TEOO. [7]

```
IF ('late return penalty' = true OR 'location penalty' == true
THEN C-ACT 'Penalty payment' [must be] requested
IF [car drop-off is considered justifiable]
THEN C-ACT 'Car drop-off' [must be] accepted
ELSE C-ACT 'car drop-off' [must be] rejected
```

Fig. 14. Excerpt from the action rule in our format

As shown on Fig. 14, our grammar allows a much easier way to understand which actions/conditions lead to calls of other transactions, such as C-ACT 'Penalty payment [must be] requested', and that same transaction type can also have an action rule with its set of actions to be carried out, which will take in account the values written/evaluated as specified. We find the current standard brings also ambiguity with the use of the *some* clause. In the example being analysed, the drop-off branch should be clearly defined by the context/instance at run time and a different specification should be done or not needed at all. It is claimed that DEMO models are supposed to be independent of implementation and/or infological/datalogical aspects. In other works we have been defending that DEMO models allow us an abstraction from reality and a reduction of complexity, but they cannot be separated from reality/implementation and action rules are the perfect spot to realize this connection. DEMO's Construction Model which has the higher level and complete view of a process as a tree of

transactions and actor roles indeed is quite abstracted from implementation. But delving onto the domain of business rules and execution, which is addressed on DEMO's Action Rules, there is a dire need for a more systematic and simple connection to reality/implementation. Current use of *with* clauses are in fact connecting to reality/implementation with clauses such as: the requested production time of penalty payment is Now and also dealing with infological/datalogical issues with clauses like the one calculating the penalty amount, so it's only natural that we "walk the last mile" and allow the specification of implementation details in the action rule specifications, to the point of client output, database updates, external calls to other systems, etc. and still in a way that is independent of specific technology. We are in fact allowing a very detailed specification of the implementation model, as per the GSDP [5] philosophy associated with DEMO theories. This model can then be directly run (with no compilation steps) in a live system like our DISME prototype.

6 Conclusions and Future Work

As can be seen from the above discussion, the Action Rule Syntax we propose in this paper is more complete, flexible and easier to read/understand/implement/run.

Our approach is better because we clearly specify what types of action will be undertaken and what inputs or outputs will be made by the system/user, what asynchronous calls to other transactions or IS will be performed. The Action Model is the perfect place to bridge the higher level models (Construction Model and State Model) to the implementation model. As it can be seen by the Action Rules examples in TEOO, specifying action rules in an abstracted way from the implementation leads to complex and impractical rules, especially difficult to interpret due to the orientation to the claims on justice, sincerity and truth. Business analysts should be able to design action rules already thinking and designing implementation issues such as logical rules that control the flow and assigning system properties to forms for input, to logical and arithmetical expressions for evaluation and output. The practical engineering approach we are following allows that, with minimal training on certain language constructs, specialized business analysts are able to "program" the flow of their enterprise in a way that directly connects strategic high level models with low level details of implementation. There are many open ends in our current prototype like how to handle external calls to other systems in the IT environment, either for input our output and we foresee the number and complexity of our action types will for sure increase. However, the philosophy that we follow and was presented in this paper seems to be a promising approach.

References

1. Dalal, S., Chhillar, R.S.: Case studies of most common and severe types of software system failure. Int. J. Adv. Res. Comput. Sci. Softw. Eng. **2**, 341–347 (2012)
2. Shull, F., et al.: What we have learned about fighting defects. In: Proceedings of 8th International Software Metrics Symposium, pp. 249–258 (2002)

3. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure–inducing input. IEEE Trans. Softw. Eng. **28**(2), 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)
5. Dietz, J.L.G.: Enterprise Ontology: Theory and Methodology. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-33149-2
6. Dumay, M., Dietz, J.L.G., Mulder, H.: Evaluation of DEMO and the language/action perspective after 10 years of experience. In: Proceedings of LAP 2005 (2005)
7. Perinforma, A.P.C.: The Essence of Organisation: An Introduction to Enterprise Engineering. Sapio Enterprise Engineering, Leidschendam (2015)
8. Aveiro, D., Silva, A.R., Tribolet, J.: Towards a G.O.D. organization for organizational self-awareness. In: Albani, A., Dietz, Jan L.G. (eds.) CIAO! 2010. LNBIP, vol. 49, pp. 16–30. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13048-9_2
9. Aveiro, D., Silva, A.R., Tribolet, J.: Extending the design and engineering methodology for organizations with the generation operationalization and discontinuation organization. In: Winter, R., Zhao, J.Leon, Aier, S. (eds.) DESRIST 2010. LNCS, vol. 6105, pp. 226–241. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13335-0_16
10. The Open Group: ArchiMate® 2.1. http://pubs.opengroup.org/architecture/archimate2-doc/
11. Object Management Group: BPMN 2.0. http://www.omg.org/spec/BPMN/2.0/
12. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Inf. Softw. Technol. **50**, 1281–1294 (2008)
13. Ettema, R., Dietz, J.L.G.: ArchiMate and DEMO – mates to date? In: Albani, A., Barjis, J., Dietz, J.L.G. (eds.) CIAO!/EOMAS -2009. LNBIP, vol. 34, pp. 172–186. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01915-9_13
14. Aveiro, D., Pinto, D.: Universal enterprise adaptive object model. In: Presented at the 5th International Conference on Knowledge Engineering and Ontology Development (KEOD), Vilamoura, Portugal, September 2013
15. 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
16. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. Manag. Inf. Syst. Q. **28**, 75–106 (2004)
17. Hevner, A.R.: A three cycle view of design science research. Scand. J. Inf. Syst. **19**, 4 (2007)
18. Dietz, J.L.G.: Is it PHI TAO PSI or Bullshit? Presented at the Methodologies for Enterprise Engineering Symposium, Delft (2009)
19. 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
20. Dietz, J.L.G., Albani, A.: Basic notions regarding business processes and supporting information systems. Requir. Eng. **10**, 175–183 (2005). https://doi.org/10.1007/s00766-005-0002-9
21. Andrade, M., Aveiro, D., Pinto, D.: DEMO based dynamic information system modeller and executer. In: IC3K 2018 (2018)