A New Action Rule Syntax for DEmo MOdels Based Automatic worKflow procEss geneRation (DEMOBAKER)

Carlos Figueira[1] and David Aveiro[1,2,3]

[1] Exact Sciences and Engineering Centre, University of Madeira,
Caminho da Penteada 9000-105 Funchal, Portugal
[2] Center for Organizational Design and Engineering,
INESC-INOV Rua Alves Redol 9, 1000-029 Lisboa, Portugal
[3] Madeira Interactive Technologies Institute,
Caminho da Penteada 9020-105 Funchal, Portugal
`carlos.figueira.oelabuma@gmail.com, daveiro@uma.pt`

Abstract. The current way of specifying Action Rules in the Design and Engineering Methodology for Organizations (DEMO) is ambiguous and leads to incomplete specifications that do not contain enough ontological information so that we can more systematically convert DEMO models to comprehensive Business Process Management and Notation (BPMN). With our proposal we now can specify – still at an ontological level – much more needed details and essential information for a more complete and close to automatic generation of BPMN models. Action rules are also the perfect spot to already specify functional and implementation requirements for the information systems supporting the Workflow Management System running such BPMN models. Thus we also contribute to bridge the huge gap between DEMO models and important implementation issues that arise at design time and should immediately be specified together with ontological elements.

Keywords: enterprise engineering, BPMN, BPM, DEMO, meta model, action model, action rules, syntax, workflow, information systems, requirements.

1 Introduction

Most IT projects fail to meet final user's expectations. From [1], where some case studies were made, a recent 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. From these case studies, it was found that some of the common causes of software failures are: the lack of clear, well-thought-out goals and specifications, poor management and poor communication among costumers, designers and programmers [4], unrealistically low budget requests, and underestimates of time requirements, use of new technologies maybe for which the software developers don't have adequate experience and expertise and refusal to recognize or admit that a project is in trouble [1]. DEMO [5] is a renowned enterprise engineering method associated with a sound set of theories aiming to contribute to solve these serious problems. However many open ends exist. For example, the produced models are used mostly for isolated efforts of organizational analysis and providing support for discussing changes initiatives. And one of its main aspect models – the Action Model – is barely used in practice [6], although the founder of DEMO himself says it's probably the most important model and where all essential model information can be found and all the other 3 aspect models can be derived from [5].

Research presented in this paper is integrated in a wider research project aiming to develop a software platform to support collaborative and semantic web based production of organizational models and diagrams that specify organizational processes, human and software responsibilities, information flows, procedures and other kinds of organizational artifacts. Such models should consist in a continuously and collaboratively updated "picture" of the reality of an enterprise that guides its collaborators in: (1) the perception of the global "organizational self" [7] (2) the execution of their operational work and (3) the creative process of changing the organization itself [8], including or not the change or implementation of software and related information technology. Other widespread approaches such as ArchiMate [9] and BPMN [10] suffer highly from the lacking of a solid formal theory behind them and from ambiguous semantics [11] [12]. Our DEMO based approach, grounded in solid theory, aims to allow the generation of models that capture vital information of organizational responsibilities and information flows, normally neglected in other approaches. From these models – that have a very high-level of abstraction and are easy to share and comprehend – we aim to systematically derive increasingly detailed models down to runnable workflows and program code or manual work instructions. All models and all model artifacts are to be formally connected with each other in a coherent and semantically strong way, bringing immense power to our approach. Our prototype software platform is inspired on the Universal Enterprise Adaptive Object Model (UEAOM) [13] and is supported by the Semantic MediaWiki (SMW) software, to allow the integrated management and adaptation of: (1) enterprise models, (2) their representations, (3) their underlying meta-models, i.e., their abstract syntax, (4) the representation rules, i.e., the concrete syntax for the respective models, and (5) automated or semi-automated generation of: (i) runnable workflows; (ii) formal requirements for software to support such workflow and (iii) program code. All this for different modelling languages and also different versions of these languages, with an initial focus on DEMO and BPMN the most widespread standard used for workflow management systems. One of the components of our software prototype is called DEMOBAKER, having as a main goal to provide an efficient and standardized way of converting DEMO models into totally compliant BPMN models having clear semantics, an issue lacking in traditional BPMN approaches. In this paper we present an important step in this direction. Namely, we propose an improved meta-model for DEMO's Action Model also presented in the form of a new Action Rule Syntax. We focus on the problem that the current way of specifying Action Rules in DEMO is ambiguous and leads to incomplete specifications that do not contain enough ontological information so that we can convert DEMO models to comprehensive BPMN models. With our proposal we now can specify – still at an ontological level – much more needed details and essential information for a more complete and close to automatic generation of BPMN models. Action rules are also the perfect spot to already specify functional and implementation requirements for the information systems supporting the Workflow Management System running such BPMN models. Thus we also contribute to bridge the huge gap between DEMO models and important implementation issues that arise at design time and should immediately be specified together with ontological elements. We use the EU-rent case taken from [14] to exemplify and validate our contribution.

2 Research Method

According to A. R. Hevner [15] [16], Design Science Research – the Information Systems Research paradigm that we adopt – should be seen as a group of three closely related cycles of activities. These activities are depicted on Figure 1. Hevner claims that the individual application of these three activities in an isolated way does not constitute good design science research. Only the conjunction of the three can actually render good design science research with a valid output. In our research, and regarding the relevance cycle depicted on Figure 1, we identified a clear problem of ambiguity and lack of concise and essential information on current DEMO's action rule syntax. So an opportunity to devise a more sound and comprehensive syntax was at hand. Regarding the Rigor cycle, our research was supported by all the theoretical foundations grounding DEMO as well as the UEAOM patterns. The most important cycle is the Design cycle itself, out of which resulted our proposal of a new meta-model for DEMO's Action Model. An exhaustive and thorough evaluation was done with many iterations of this cycle where we would be adding new elements to DEMO's Action Meta Model and instantiating our new syntax with the EU-rent case and evaluating if it allowed to specify maximum ontological information in a concise and comprehensive way, normally not the case in DEMO's current standard Action Models. While instantiating and increasing the complexity of EU-rent case's action rules to a more realistic level, some times we found some concept in the meta-model should be unary, some other times other concepts should be binary, and, at other times, we found we would need to specify new concepts at meta-level like: *atomic action* and *flow*. So the proposal presented in this paper is the result of a long and thorough process of conceptual evolution and comprehensive instantiation, thus following the tenets of Design Science Research.

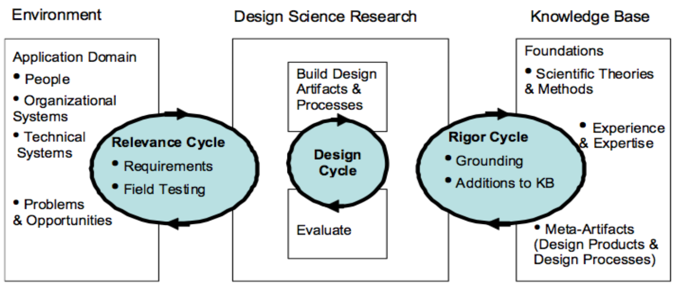


Fig. 1. Design science research cycles [17]

3 Background and Theoretical Foundations

3.1 DEMO's Operation, Transaction and Distinction Axioms

In the Ψ-theory [17] – 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 Figure 2. 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 [18].

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 Figure 3, 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 [18] 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".

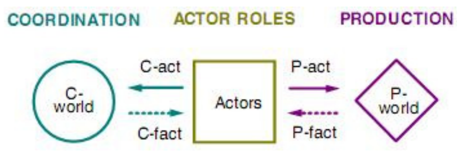


Fig. 2. Actor's Interaction with Production and Coordination Worlds [5]

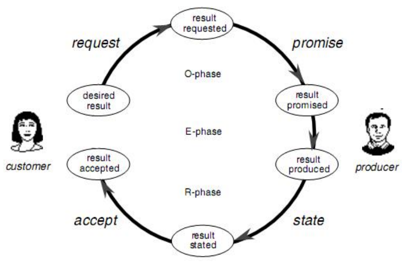


Fig. 3. Basic Transaction Pattern [5]

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 [18].

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 [18]. 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. And 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 [19].

3.2 Business Process Management (BPM) and Its Notation – BPMN

As stated in [20], “Business Process Management (BPM) is the discipline that describes structured methods and techniques used to make a business process more efficient adaptive and effective for accomplishing a specific task within an organization” BPM techniques and methods also allow the identification and modification of existing processes in order to align them to future possibilities of change. BPMN, stands for Business Process Model and Notation and consists in a method for graphically representing the steps of a business process similar to a flowchart approach. The BPMN notation was specifically designed to allow the specification of the coordination of the sequence of organizational processes and the way that messages flow between activities, processes and participants.

3.3 Universal Enterprise Adaptive Object Model

The Universal Enterprise Adaptive Object Model (UEAOM) is a recent proposal from [13], consisting in a conceptual schema inspired in the the Adaptive Object Model (AOM) [21], a software architecture pattern for systems in which classes, attributes, relationships and behaviors of applications are represented as metadata, allowing them to be changed in runtime environment.

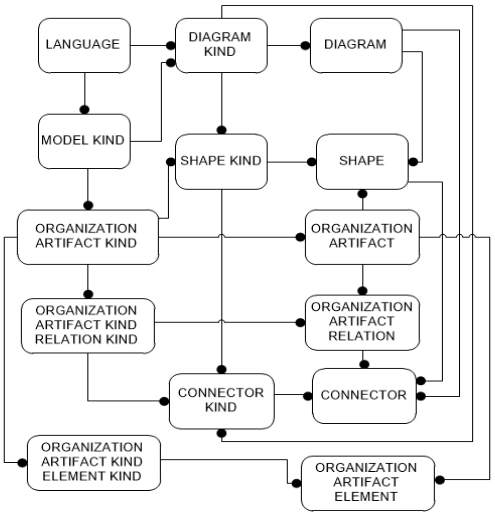

LANGUAGE
DIAGRAM KIND
DIAGRAM
MODEL KIND
SHAPE KIND
SHAPE
ORGANIZATION ARTIFACT KIND
ORGANIZATION ARTIFACT
ORGANIZATION ARTIFACT KIND RELATION KIND
ORGANIZATION ARTIFACT RELATION
CONNECTOR KIND
CONNECTOR
ORGANIZATION ARTIFACT KIND ELEMENT KIND
ORGANIZATION ARTIFACT ELEMENT

Fig. 4. UEAOM - simplified version

Figure 4 presents a simplified version of the original UEAOM, with the core classes only. A brief explanation and exemplification of the core classes relevant for the contributions of this paper follows. LANGUAGE – used to specify which languages are permitted in the diagram editor. Example: «DEMO v3.5». MODEL KIND – each language can have multiple model kinds, used to specify which kinds of models are permitted for a certain language in the Diagram Editor. Example: «Construction Model v3.5». DIAGRAM KIND – each model can have multiple Diagram kinds, used to specify which kind of Diagrams are permitted in the Diagram Editor for a certain model kind. Example: The Actor Transaction Diagram «ATD v3.5». ORGANIZATION ARTIFACT KIND (OAK) – used to specify which kinds of organization artifacts can be used in models. Example: «ELEMENTARY ACTOR ROLE v3.5». ORGANIZATION ARTIFACT KIND RELATION KIND (OAKRK) – used to specify which kinds of relations are permitted between OAKs. Example: «TRANSACTION KIND.executed by.ELEMENTARY ACTOR ROLE v3.5». ORGANIZATION ARTIFACT (OA) – used to specify a concrete organization artifact instance of a particular OAK. Example: «A01 - rental starter» is an instance of ELEMENTARY ACTOR ROLE v3.5. ORGANIZATION ARTIFACT RELATION (OAR) – used to specify a concrete organization artifact relation instance of a certain OAKRK. Example: «CAR PICK UP.executed by.RENTAL STARTER». ORGANIZATION ARTIFACT KIND ELEMENT KIND (OAKEK) – used to specify an element kind of an organization artifact kind. Example: Transaction Name is an element of the Organization Artifact Kind Transaction. ORGANIZATION ARTIFACT ELEMENT (OAE) – used to specify instances of an OAKEK. Example: the string "rental start" hat is the name given to a particular transaction that is an instance of the Organization Artifact Kind Transaction.

4 From DEMO to BPMN Workflow with Precise Semantics

The major goal of the DEMOBAKER project is to allow the conversion of UEAOM based DEMO models into compliant UEAOM based BPMN processes. Looking at DEMO models, we concluded that Action Rules would be the main source of information for this conversion process, as they specify, for all transactions, all agendum for each of the internal actors of the organization, that is all coordination facts that they have to respond to and then which conditions have to be verified, facts created, etc. After several experiments of converting DEMO models to BPMN models we found that we had to discard several ambiguous elements from BPMN – e.g., the message element that easily becomes redundant with tasks – and arrived at the following conversion rules – one of the main contributions of this paper – so that each DEMO concept has a 1 to 1 correspondence to a BPMN concept. We use the format: **DEMO concept < > BPMN concept**. Because DEMO has a strong semantics with a comprehensive meta-model, these proposed rules imply that, by using the few BPMN elements we select, we have a more precise semantics for BPMN compared to an unrestricted use of it or to using BPMN as a starting point to model enterprise processes.

Transaction < > Pool – each transaction is represented on BPMN as a Pool, and inside of this pool we can only have its related coordination and production acts/facts. Each transaction must start with a *start event* and finish with an *end event*. The start event must be triggered by another transaction. The end event can occur due to several reasons, for example, due to a revoke request act realized by a certain actor.

Actors < > Lane – Actors initiate and/or execute transactions, so each transaction has two lanes, one for the initiator and another for the executor and all events depicted inside a lane are of the responsibility of the respective actor.

Flows and Conditions < > Tasks – Flows, conditions and their evaluations (in action rules) are all represented on BPMN as tasks. Tasks have an input and an output flow. Each actor is responsible for the tasks depicted in their respective lane.

Coordination-Facts/Production-Facts < > Signals – Coordination and production facts correspond to signals, meaning that they are used as throw and catch signals. This is a key conversion rule and a very important contribution of our research. In this manner we can "isolate" the specification of each transaction and their respective rules in a pool and the occurrence of certain facts will possibly enact one or more transactions at the same time. This provides a high degree of flexibility, modularization and paralelism.

5 New DEMO Action Rule Syntax

As already mentioned, to present our proposal of a new DEMO Action Rules Syntax, we use the case EU-rent from [14]. In Figure 5 we find one of the action rules from this case. And this one is a perfect example on how Action Rules are “the neglected son” of DEMO. In real life, many different conditions and facts have to be verified before one can proceed to accept the drop-off of a car. In this action rule specification, the only verified fact is if the branch where the car is delivered is the same as the contracted one. But no action is specified for the case it is not. Also, in this rule we find a common problem in DEMO's Action Rule Meta-Model: what is the meaning of the construct *with*? We find it in many action rules and, apparently, with different functions: creating new facts, verifying new facts, etc. Indeed, in the most current public version of DEMO, it is assumed that “the syntax and the formal semantics of the action rules need to be elaborated yet” [22]. We agree that it is valuable to have models that abstract from infological and datalogical aspects as well as implementation issues, like DEMO's Construction Model, Process Model and State Model do. However, DEMO models can never be fully independent of implementation and resource constraints arising from the organization's reality. Rather, at most, they are implementation abstracted [23]. We claim that Action Rules are the perfect spot to make the bridge between the implementation world and the most implementation abstracted views of an organization – like that of transactions and actor roles of the Actor Transaction Diagram. We claim this because, while thinking on the flow and requirements for the action of actor roles, we inevitably need to think about necessary fact evaluations, information requests, data storing etc. So why not specify immediately such items while devising DEMO action rules? In this manner, the ontological model of an organization fully guides the specification of relevant infological, datalogical actions, as well as implementation requirements. We present, in Figure 6, the action rule that is the final result of the evolution of the simple action rule from Figure 5. This rule is already structured according to the new syntax we propose and is the result of several iterations of the next steps – that follow design science research method mentioned in Section 2: (1) devising new meta-model constructs for DEMO's Action Model and (2) evaluating their applicability and comprehensiveness by instantiating all the action rules of the EU-rent case. For step 1 we would create instances of UEAOM classes OAK, OAKRK and OAKEK, thus, specifying the meta-model. For step 2 we would create instances of UEAOM classes OA, OAR and OAE, that is, creating an Action Model following the specified meta-model. For space reasons we present only the action rule of Figure 6, which is already enough to explain and justify our proposal. This rule example shows many elements necessary for BPMN specification. The BPMN model resulting from our conversion rules – shown in Figure 7 – depicts the steps of the respective transaction and different kinds of infological and datalogical actions while still being totally abstracted from the implementation. The already mentioned design science research cycle of creation of, both the UEAOM instantiation for the meta-model specification and the instantiation for the full action rule specification of the EU-rent case, continued, until an instantiation gathered in a comprehensive way all the necessary information to have a real-life BPMN process fully specified, excluding implementation details. In our BPMN example we find all needed information, namely, all relevant ontological, infological and datalogical steps regarding the process of dropping the car at a branch.

WHEN	car drop-of of [rental] is stated
with	the actual drop-off branch of [rental] is [branch]
then	car drop-of of [rental] must be accepted

Fig. 5. Action Rule example

Figure 8 shows the UEAOM instantiation that consists in the new Action Meta-Model we propose and the explanation of each of its elements is now due. Due to the UEAOM following the AOM pattern and also the type square pattern, the explanation that follows could appear confusing, having too many instantiations. But the reader just needs to keep in mind that these patterns provide immense power of adaptability to systems in runtime due to the fact that instances of classes of an AOM may be themselves types or "classes" and that instances of AOM classes may, in turn, have instance kind relationships between them.

The first instance of the UEAOM class OAK that we need is, of course, the one specifying the *action rule* concept type itself. It has an associated OAKEK instance for its identification that we call *action rule id*. In our example, an instance of the UEAOM class OA would be action rule *AR05*. Being that the AR05 string or value is an instance of an OAE, itself instance of the OAKEK action rule id. This rule AR05 is, itself, an instance – at model level – of the just before mentioned OAK instance action rule – in turn, at meta-model level. To the reader that had no knowledge of the type square pattern. These several "double instantiations" described in the previous sentences constitute an example of 4 AOM instances/objects, forming such a type square, where, on one side have a type and a property type (part of the type) and on the other side of the square we have an instance of the type containing a value, itself instance of the respective property.

AR05						
WHEN	car drop-off of [rental] is stated	C-FACT	ARCW01 (ARCET01)			
ACTION (FIRST, ATOMIC)	Read contracted car drop-off details	READ_DATA	ARCA01			
ACTION (FLOW)	**if** the actual drop-off branch of [rental] is [contracted_branch]	fact evaluation	ARCA02 (ARCIF01(ARCC01))			
	then ACTION (FLOW)	**if** the actual date is [contracted drop-off date]	fact evaluation	ARCT01 (ARCA03(ARCIF02(ARCC02)))		
		then ACTION (FIRST, BLOCK)	ARCT02(ARCA04)			
		ACTION (AFTER, ATOMIC)	penalty charge == false	WRITE_DATA	ARCA05	
		ACTION (AFTER, ATOMIC)	car drop-off of [rental] must be accepted	C-ACT	ARCA06	
		else ACTION (FIRST, BLOCK)	ARCE01(ARCA07)			
		ACTION (FIRST, ATOMIC)	penalty charge == true	WRITE_DATA	ARCA08	
		ACTION (AFTER, ATOMIC)	late return penalty charge == true	WRITE_DATA	ARCA09	
		ACTION (AFTER, ATOMIC)	car drop-off of [rental] must be accepted	C-ACT	ARCA10	
	else ACTION (FLOW)	**if** client accept to delivery it anyway	condition evaluation	ARCE02 (ARCA11(ARCIF03(ARCC03)))		
		then ACTION (FLOW)	**if** the actual date is [contracted drop-off date]	fact evaluation	ARCT03(ARCA12(ARCIF04(ARCC02)))	
			then ACTION (FIRST, BLOCK)	ARCT04 (ARCA13)		
			ACTION (FIRST, ATOMIC)	penalty charge == true	WRITE_DATA	ARCA14
			ACTION (AFTER, ATOMIC)	location penalty charge == true	WRITE_DATA	ARCA15
			ACTION (AFTER, ATOMIC)	car drop-off of [rental] must be accepted	C-ACT	ARCA16
			else ACTION (FIRST, BLOCK)	ARCE03(ARCA17)		
			ACTION (FIRST, ATOMIC)	penalty charge == true	WRITE_DATA	ARCA18
			ACTION (AFTER, ATOMIC)	location penalty charge == true	WRITE_DATA	ARCA19
			ACTION (AFTER, ATOMIC)	late return penalty charge == true	WRITE_DATA	ARCA20
			ACTION (AFTER, ATOMIC)	car drop-off of [rental] must be accepted	C-ACT	ARCA21
		else ACTION (FIRST, ATOMIC)	car drop-off of [rental] must be rejected	C-ACT	ARCE04(ARCA22)	

Fig. 6. Action Rule example following the new syntax

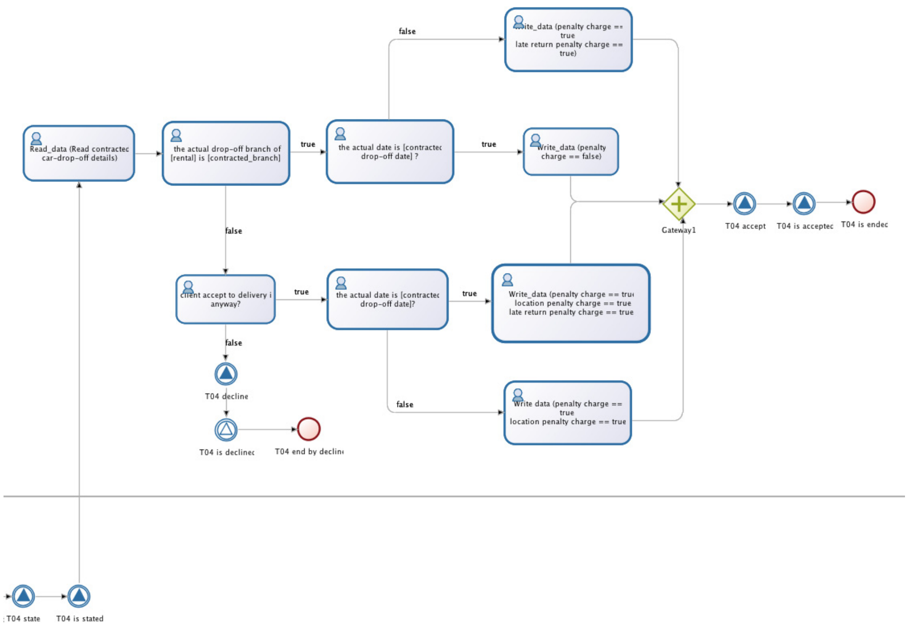
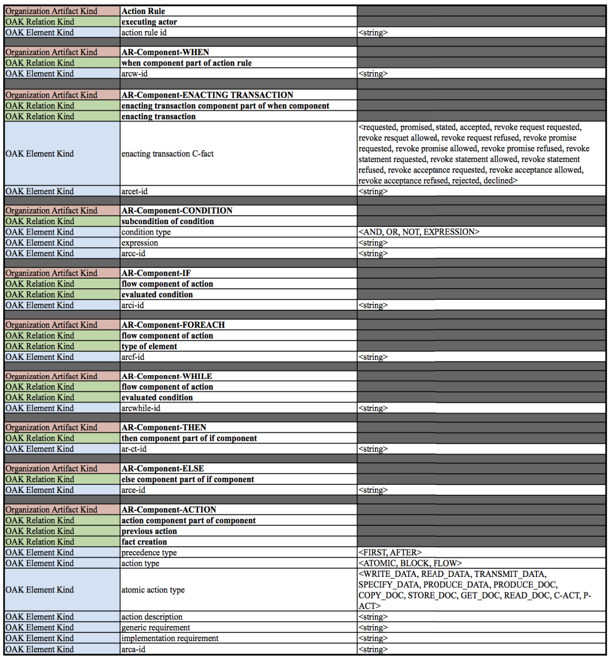

Read_data (Read contractec car-drop-off details)
the actual drop-off branch of [rental] is [contracted_branch]
true
the actual date is [contractec drop-off date] ?
false
Write_data (penalty charge == true late return penalty charge == true)
true
Write_data (penalty charge == false)
Gateway1
T04 accept
T04 is acceptec
T04 is endec
false
client accept to delivery i anyway?
true
the actual date is [contractec drop-off date]?
true
Write_data (penalty charge == true location penalty charge == true late return penalty charge == true
false
T04 decline
T04 is declinec
T04 end by declin
false
Write data (penalty charge == true location penalty charge == true
T04 state
T04 is stated

Fig. 7. BPMN diagram for Transaction T04 - car drop off



Organization Artifact Kind	**Action Rule**	
OAK Relation Kind	**executing actor**	
OAK Element Kind	action rule id	<string>
Organization Artifact Kind	**AR-Component-WHEN**	
OAK Relation Kind	**when component part of action rule**	
OAK Element Kind	arcw-id	<string>
Organization Artifact Kind	**AR-Component-ENACTING TRANSACTION**	
OAK Relation Kind	**enacting transaction component part of when component**	
OAK Relation Kind	**enacting transaction**	
OAK Element Kind	enacting transaction C-fact	<requested, promised, stated, accepted, revoke request requested, revoke resquet 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>
OAK Element Kind	arcet-id	<string>
Organization Artifact Kind	**AR-Component-CONDITION**	
OAK Relation Kind	**subcondition of condition**	
OAK Element Kind	condition type	<AND, OR, NOT, EXPRESSION>
OAK Element Kind	expression	<string>
OAK Element Kind	arcc-id	<string>
Organization Artifact Kind	**AR-Component-IF**	
OAK Relation Kind	**flow component of action**	
OAK Relation Kind	**evaluated condition**	
OAK Element Kind	arci-id	<string>
Organization Artifact Kind	**AR-Component-FOREACH**	
OAK Relation Kind	**flow component of action**	
OAK Relation Kind	**type of element**	
OAK Element Kind	arcf-id	<string>
Organization Artifact Kind	**AR-Component-WHILE**	
OAK Relation Kind	**flow component of action**	
OAK Relation Kind	**evaluated condition**	
OAK Element Kind	arcwhile-id	<string>
Organization Artifact Kind	**AR-Component-THEN**	
OAK Relation Kind	**then component part of if component**	
OAK Element Kind	ar-ct-id	<string>
Organization Artifact Kind	**AR-Component-ELSE**	
OAK Relation Kind	**else component part of if component**	
OAK Element Kind	arce-id	<string>
Organization Artifact Kind	**AR-Component-ACTION**	
OAK Relation Kind	**action component part of component**	
OAK Relation Kind	**previous action**	
OAK Relation Kind	**fact creation**	
OAK Element Kind	precedence type	<FIRST, AFTER>
OAK Element Kind	action type	<ATOMIC, BLOCK, FLOW>
OAK Element Kind	atomic action type	<WRITE_DATA, READ_DATA, TRANSMIT_DATA, SPECIFY_DATA, PRODUCE_DATA, PRODUCE_DOC, COPY_DOC, STORE_DOC, GET_DOC, READ_DOC, C-ACT, P-ACT>
OAK Element Kind	action description	<string>
OAK Element Kind	generic requirement	<string>
OAK Element Kind	implementation requirement	<string>
OAK Element Kind	arca-id	<string>

Fig. 8. UEAOM based Action Rule Meta-Model specification

Next, we needed to specify which actor role has the responsibility of executing this rule. Thus we needed to specify an instance of class OAKRK, that we call *executing actor*, relating the action rule OAK with the actor role OAK. In our example, an instance of an OAR that would be an instance at model level of the OAKRK executing actor (in turn, at meta-model level) would be the the OAR relating AR05 with actor role A01, named rental starter. We also needed to specify the fact that triggers an action rule. For that we specified the OAK *AR-Component-When*, also needing an identifier, specified as the OAKEK *arcw-id*. In our example the instance is ARCW01. Next, we need to specify to which action rule this AR-Component-When is related to. Thus the specification of the following OAKRK: *when component part of action rule*. An instance of an OAR in our example would be the one relating ARCW01 to AR05. The OAK *AR-Component-Enacting-Transaction* serves to specify facts that trigger an action rule. In our example, an OA instance that is an instance of this OAK instance is: ARCET01. This complexity is needed as an action rule can be triggered by one or more c-facts of different transactions. The OAKRK *enacting transaction component part of when component* serves to relate the previous OAK with the *when component* OAK. In our example, an instance of an OAR instance of this OAKRK would be the OAR relating ARCET01 with ARCW01. The OAK Relation Kind *"enacting transaction"* specifies the relation between the OAK *AR-Component-Enacting-Transaction* and the OAK transaction. In our example, an instance of an OAR instance of this OAKRK would be the OAR relating ARCET01 with transaction T04 named car drop-off. The OAKEK *"enacting transaction C-Fact"* serves to specify which fact (out of the possible 21 facts of the universal transaction pattern) of the related transaction triggers the execution of this action rule. In our example we have the OAE *stated*. All the above instances of classes OAK, OAKRK and OAKEK are needed to precisely specify, at meta-model level, the structure of the initial part of an action rule. The above mentioned instances of classes OA, OAR and OAE specify at model level the following part of our example action rule: *"when car drop-off of [rental] is stated.*

Similar reasonings were followed for all other OAKs, OAKRKs and OAKEKs that we specify and are presented in Figure 8. We now only justify their specification. We need to specify the OAK *condition*. Conditions are evaluated by what we called *Flows* (If, While), explained later. A condition has to be of a certain (OAKEK) *condition type*, namely: *AND*, *OR*, *NOT*, or *EXPRESSION*. In the first three cases the condition is actually a composite condition where we have a boolean operator followed by a (OAKRK) sub-condition. In the last case the condition will actually be an atomic condition in the form of a boolean expression. In our example in Figure 6, we would have, as instance of class OA, the atomic condition: ARCC01 with expression *the actual drop-off branch of [rental] is [contracted_branch]*. In the case we have composite conditions (in fact a tree of conditions) each non-root condition needs to specify that they are *sub-condition of condition*. Imagining we would have a NOT condition with the previous example being the expression, we would have an OAR specifying that ARCC02 is a sub-condition of ARCC01, where the condition type OAE associated with the OA ARSC02 would be: NOT. Having the conditions specified we need to specify in which context they are evaluated. This happens in two kinds of *flow* component. Both the (OAK) *if* and the (OAK) *while* components evaluate a condition specified by the (OAKRK) *evaluated condition*. We specify to which action rule they belong thanks to OAKRK *flow component of action*. In our example we would have the OAR: ARCIF01 evaluating the condition ARCC01. The *foreach* flow has an action for each of its *type of element*. An if component is composed by its respective (OAK) *then* and (OAK) *else* components. In our example: the OAK ARCT01 being the *then component part of if component* ARCIF01 and the OAK ARCE01 as the *else component part of if component* ARCIF01. Finally we needed to specify actions of the action rule itself, thus the OAK AR-Component-Action. Actions always belong to some component, relation specified by OAKRK *action component part of component*. In our example: There is an OAR specifying the OA ARCA01 as being the *action component part of component* OA ARCW01 (the when component of action rule AR05). An action component can be followed by another action and such precedences are specified by the OAKRK *previous action*. In this manner we can "program" actions sequentially. For instance and based on our example: The OA ARCA05 that has as *previous action* the OA ARCA04. When an actor will perform an action, some fact can be created and that relation is specified by the OAKRK *fact creation*. Since actions can proceed other actions or may be the first action for a certain component, we had the need to be able to specify this differentiation. Thus we specified the OAKEK *precedence type* where the allowed values for the OAE are strings *FIRST* or *AFTER*. To specify if the action component in question is an atomic action or a block of actions or a flow we need OAKEK *action type*. In our example the OA ARCA04 is the FIRST action of the then component and is of type BLOCK, that is, it just specifies that we will have a block of actions executed. The OA ARCA05 is the FIRST ATOMIC action of the action block ARCA04. An atomic action can be of many types and this is specified by OAKEK *atomic action type*. As we can see in Figure 8 there are datalogical, infological and ontological acts as options. In our example: the OA ARCA05 has its OAE with value *WRITE_DATA*. OAKEK *action description* serves to specify/describe in a formal/informal way the atomic action in question. In our example we have the OAE *penalty charge == false*. An action can also have specific requirements still abstracted from implementation or implementation-dependent. Thus the OAKEKs *generic requirement* and *implementation requirement*. As an example of a generic requirement we could specify that a certain action of type PRODUCE_DATA "is mandatory" which means that the actor has to really get or produce such data from somewhere. An example of an implementation requirement would be "obtain record from the government driving license system's web service"

We could produce an OFD (the DEMO option for specifying meta-models) for another alternative view (other than the UEAOM instantiation) of our newly proposed meta-model for DEMO Action Rules. Due to lack of space and taking in account that we are kind of specifying what we could call the *organization programming language* we choose to present, in Figure 9, our newly proposed syntax using the Backus-Naur Form (BNF) notation, which summarizes our main contribution of this paper.

<When> : : =	<Transaction kind name> "of" <Object Identifier> "is" <C-Fact> | <P-Fact> <Action> | <Flow>
<Transaction kind name> : : =	<String>
<Object Identifier> : : =	"[" <String> "]"
<P-Fact> : : =	"produced"
<C-Fact> : : =	"requested" | "promised" | "stated" | "accepted" | "revoke request requested" | "revoke resquet 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"
<C-Act> : : =	<Transaction kind name> "of" <Object Identifier> "must be" <C-Fact>
<P-Act> : : =	<Transaction kind name> "of" <Object Identifier> "must be" <P-Fact>
<Flow> : : =	<Foreach> | <If> | <While>
<Action> : : =	<Atomic action> | <Block> | <Flow>
<Atomic action> : : =	<Action type> <Action description> | <Generic requirement> | <Implementation requirement>
<Action description> : : =	<String>
<Generic requirement> : : =	<String>
<Implementation requirement> : : =	<String>
<Action type> : : =	"Write_Data" | "Read_Data" | "Transmit_Data" | "Specify_Data" | "Produce_Data" | "Produce_Doc" | "Copy_Doc" | "Store_Doc" | "Get_Doc" | "Read_Doc" | <C-Act> | <P-Act>
<Block> : : =	<Action> { <Action> }
<Condition> : : =	<Condition type> { <Condition> } | <Expression>
<Condition type> : : =	"And" | "Or" | "Not" | <Expression>
<Expression> : : =	<String>
<If> : : =	"If" <Condition> "then" <Action> | <Flow> "else" <Action> | <Flow>
<While> : : =	"While" <Condition> <Action> | <Flow>
<Foreach> : : =	"Foreach" <type of element> <Action> | <Flow>

Fig. 9. New Action Rule Syntax

6 Discussion, Conclusions and Future Work

When one really starts to specify details, still at infological and datalogical levels, things can get really complex. By looking at the full model of the EU-Rent case [14], one can see that it's only on the rule that handles the promise of the rental end transaction, that the payment transaction is requested (possibly with incurring fines). Imagine that the renter would drop off the car by mistake in a branch different from the contracted one. After dropping off the car at the garage he or she would get quite a surprise, at the branch desk, when having to pay the fine. It makes much more sense that the action rule that handles the state c-fact of the car drop-off transaction evaluates if the drop-off branch is correct and already informs the renter of the fine he would have to pay if he wishes to proceed. This gives him or her a chance of not proceeding and maybe leave this branch and deliver the car on the correct branch. Just this example shows that, while modeling DEMO transactions, one should already have in mind implementation issues that will affect transaction design. Depending if (1) there is a garage attendant and then the rental desk handles the payment or if (2) there is just the garage attendant himself which takes care of both the car drop-off and penalty payments; this will have a profound impact on the design of the action rules and maybe even on transaction design itself. In this later case one could “fuse” the current car drop-off and rental end transactions. In the more complete action rule we present on this paper, we see that an apparently simple rule was in fact “forgetting” lots of complexity in terms of conditions to be verified and the creation of many original facts (e.g., flags regarding the fines) that are themselves, very relevant actions to be correctly and comprehensively implemented in a BPMN flow in the correct spot of the flow, like we depict in Figure 7. We started off with the purpose of automatically generating runnable BPMN models from DEMO models. With this end in mind, we had first to solve the problem of lack of semantics in BPMN. Hence, we selected a few BPMN concepts (out of the many ambiguous ones available) and assigned to them clear and precise semantics thanks to the the set of 1 to 1 conversion rules from DEMO to BPMN that we propose. But in the middle of the process we found yet another problem: the main information source in DEMO to generate BPMN flows – the Action Rules Specification of the Action Model – was not precise enough nor had clear semantics itself. We then took the endeavor of, following Design Science Research tenets, solving that problem by applying the Universal Enterprise Adaptive Object Model to specify a more complete and comprehensive Meta-Model, i.e., abstract syntax, for the Action Rules. After several instantiations of our ideas and evaluation of their applicability by the instantiation of all EU-rent case action rules with each new version of our Meta-Model, we kept on improving it to the stage as presented in this paper. Some very relevant contributions of the new syntax are that we are kind of finding the primitives for what we call *organization programming language*, still at an implementation abstracted level, in a way that we can produce much more comprehensive BPMN models, more close to being ready to be runnable, compared to other existing approaches. The current version of the syntax has several aspects to improve still. Namely when we have an atomic action that is a C-ACT, we must have an additional OAKEK to specify to which transaction this C-ACT corresponds, so that the Workflow engine can throw the right signal to activate the right BPMN pool. In fact one of the next lines of future work, besides polishing up missing details in our new syntax, is the creation of a parser that takes, as an input, action rules following our proposed syntax and outputs a runnable BPMN workflow that can be automatically imported to a well-known Open Source Workflow System where we can easily implement (or connect to) the needed database tables, queries and web forms for data production/input, following the requirements formally or informally specified in each atomic action of the action rules.

We also have one small contribution to the formal specification of the Universal Transaction Pattern. All coordination acts need to have different names. For example, in the current pattern, revoking a request is different from revoking a promise. So the allow and refuse acts – which in the current pattern have the same name for the 4 stanard c-acts – should also be differentiated according to if they correspond to allowing the revoke of a request or of a promise, for example. All the new names of c-acts we propose can be found on our new syntax and the official transaction pattern should have the names of these same acts changed accordingly.

References

1. Dalal, S., Chhillar, R.S.: Case Studies of Most Common and Severe Types of Software System Failure. International Journal of Advanced Research in Computer Science and Software Engineering 2 (2012)
2. Shull, F., Basili, V., Boehm, B., Brown, A.W., Costa, P., Lindvall, M., Port, D., Rus, I., Tesoriero, R., Zelkowitz, M.: 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 Transactions on Software Engineering (2002)
4. Dalal, S., Chhillar, R.S.: Role of Fault Reporting in Existing Software Industry. CiiT International Journal of Software Engineering and Technology (July 2012)
5. Dietz, J.L.G.: Enterprise Ontology: Theory and Methodology. Springer, Heidelberg (2006)
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. Aveiro, D., Silva, A.R., Tribolet, J.: Towards a G.O.D. Organization for Organizational Self-Awareness. In: Albani, A., Dietz, J.L.G. (eds.) CIAO! 2010. LNBIP, vol. 49, pp. 16–30. Springer, Heidelberg (2010)
8. Aveiro, D., Rito Silva, A.: Extending the Design and Engineering Methodology for Organizations with the Generation Operationalization and Discontinuation Organization. In: Winter, R., Zhao, J.L., Aier, S. (eds.) DESRIST 2010. LNCS, vol. 6105, pp. 226–241. Springer, Heidelberg (2010)
9. The Open Group: ArchiMate® 2.1 Specification, `http://pubs.opengroup.org/architecture/archimate2-doc/`
10. Object Management Group: BPMN 2.0, `http://www.omg.org/spec/BPMN/2.0/`
11. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Information and Software Technology 50, 1281–1294 (2008)
12. Ettema, R., Dietz, J.L.G.: ArchiMate and DEMO – Mates to Date? In: Albani, A., Barjis, J., Dietz, J.L.G. (eds.) CIAO! 2009. LNBIP, vol. 34, pp. 172–186. Springer, Heidelberg (2009)
13. Aveiro, D., Pinto, D.: Universal Enterprise Adaptive Object Model. Presented at the 5th International Conference on Knowledge Engineering and Ontology Development (KEOD), Vilamoura, Portugal (September 2013)
14. Jan, L.G.: Dietz: DEMO 3 - Way of Working (2009)
15. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. Management Information Systems Quarterly 28, 75–106 (2004)
16. Hevner, A.: A Three Cycle View of Design Science Research. Scandinavian Journal of Information Systems 19 (2007)
17. Dietz, J.L.G.: Is it PHI TAO PSI or Bullshit? Presented at the Methodologies for Enterprise Engineering Symposium, Delft (2009)
18. Dietz, J.L.G.: On the Nature of Business Rules. In: Dietz, J.L.G., Albani, A., Barjis, J. (eds.) CIAO! 2008 and EOMAS 2008. LNBIP, vol. 10, pp. 1–15. Springer, Heidelberg (2008)
19. Dietz, J.L.G., Albani, A.: Basic notions regarding business processes and supporting information systems. Requirements Eng. 10, 175–183 (2005)
20. BPM Tutorial - TechTarget, `http://searchsoa.techtarget.com/tutorial/BPM-Tutorial`
21. Yoder, J.W., Balaguer, F., Johnson, R.: Architecture and design of adaptive object-models. SIGPLAN Not. 36, 50–60 (2001)
22. Dietz, J.L.G.: DEMO-3 Models and Representations (2009), `http://www.demo.nl`
23. Pombinho, J., Aveiro, D., Tribolet, J.: The Role of Value-Oriented IT Demand Management on Business/IT Alignment: The Case of ZON Multimedia. In: Harmsen, F., Proper, H.A. (eds.) PRET 2013. LNBIP, vol. 151, pp. 46–60. Springer, Heidelberg (2013)