A Loose Coupling Approach for Combining OWL Ontologies and Business Rules

Amina Chniti,[1,2] Patrick Albert,[1] Jean Charlet[2,3]

[1] CAS France, IBM
{amina.chniti,albertpa}@fr.ibm.com
[2] INSERM UMRS 872, Eq 20, 15, Rue de l'école de médecine, 75006, Paris, France
Jean.Charlet@upmc.fr
[3] AP-HP, Paris, France

Abstract. In this demonstration we will show two prototypes based on the BRMS (Business Rule Management System) WODM, (1) The OWL plug-in and (2) the change-management plug-in. The OWL plug-in enables authoring and executing business rules over OWL ontologies. It consists of importing OWL ontologies into WODM and using the all the functionalities offered by this BRMS to author and execute rules. The change-management plug-in enables the evolution of business rules with respect to the ontology changes. This component, implemented basically using an OWL ontology and rules, detects inconsistencies that could be caused by an ontology evolution and proposes solution(s), called repair, to resolve them.
Keywords: Ontology, Business Rule, Consistency maintenance, inconsistency.

1 Challenges faced

In the majority of BRMS, Business Models are represented using Object Models while OWL Ontologies offer a better power of expressiveness. The purpose of the proposed challenge is to bring the expressiveness of OWL ontologies to business users by means of business rules authored in a natural controlled language. For this, we exploited the infrastructure offered by the BRMS WebSphere Operational Decision Management (WODM) and developed two prototypes : The OWL Plug-in and the change-management plug-in for WODM. the OWL plug-in enables authoring and executing business rules over OWL ontologies. It consists of importing OWL ontologies into WODM and using the all the functionalities offered by this BRMS to author and execute rules. The change-management plug-in enables the evolution of business rules authored over OWL ontology with respect to the ontology changes. This component, implemented basically using OWL ontology and rules, detects rules inconsistencies that could be caused by an ontology change and proposes solution(s), called repair, to resolve them.

During the development of this work we have been faced to the following challenges :

– how to enable business users to deal with their business knowledge formalized using OWL Ontologies?
– How to import OWL ontology into WODM?
– WODM is object model based, how to import the expressiveness and the reasoning capacity of OWL into such a BRMS?
– How to minimize the loose of information?

Ontologies evolve during their life cycle :
– What is the impact of such evolution on the rules?
– How to make the rule set evolving with respect to the ontology?
– How to maintain its consistency while it evolves?
– How to detect the impact of the ontology evolution on rules?

2 Method

In this section, we will describe the methods we developed to resolve the challenges described above. These methods are based on WODM.

WODM offers an infrastructure that enables business users to author - in a controlled natural language - execute and mange business rules in a collaborative way. As the majority of BRMS, it uses an object oriented models to formalize the domain knowledge. In WODM, this object oriented model is called BOM (Business Object Model). The BOM represents the entities of a given business (*e.g. Client*, *age*). It is generated over from the XOM (eXecutable Object Model) then verbalized. The XOM is the model enabling the execution of rules. It references the application objects and data, and is the base implementation of the BOM. The XOM can be built from compiled Java classes (Java execution object model) or XML Schema (dynamic execution object model). The verbalization of the BOM consists of generating a controlled natural language vocabulary (VOC) which enables to edit the business rules. The VOC, add a layer of terminology on top of the BOM (*e.g. "the client"*, *"the age of the client"*). This vocabulary is used to compose the text of the rules.

2.1 OWL plug-in

To enable business users to author business rules over OWL ontologies, we developed the WODM OWL plug-in. This plug-in exploits infrastructure offered by WODM to import OWL ontologies within it. The main component for authoring rules in WODM is the BOM. For this, we performed a mapping of OWL concepts (TBox) into the BOM. Thus, when we import an OWL ontology within WODM, the BOM is automatically generated and the functionalities offered by the BRMS can be used [1].

Among these functionalities, we will focus on authoring and executing rules. Once we have the BOM, its verbalization is also available and the business users are able to edit the business rules in a natural controlled language or using the decision tables or the decision tree.

To execute business rules authored over ontologies, we performed a second mapping of OWL/BOM entities to a XOM using Jena . Jena is a Java framework, including an ontology API for handling OWL ontologies, which allows to generate Java objects from the entities of the ontology. These Java objects then constitute the XOM. The use of Jena provides an execution layer for the OWL ontologies. This execution layer provides inference mechanisms on this model and the mapping of OWL concepts, properties, and individuals to a Java object model. When the business user launches the rule execution process, the ontology individuals (ABox) are loaded into the working memory of the rule engine and mapped into java objects. The rule engine evaluates the rule conditions against these objects and fires the rules for the objects that meet the condition. During this process, the ABox mapped into Java Objects, is updated with respect to the action parts of the fired rules. At the end of the execution process, the "new" ABox is loaded into the ontology.

Another important point in the process of executing rules is the interaction between the classification engine and the rule engine. In the following we will present some examples of this kind of interaction as achieved with the system presented in this work. The classification engine assigns the type of the individuals, then the rule engine uses this inferred knowledge to trigger a computation that could not be easily represented in an ontology. In other words, the rule engine asks the classification engine for the type of the individual, then it executes the rule(s) matching with the returned type.

Example 1: In the ontology, we define the concept **CarDriver** as a **Driver** who has a **CarDrivingLicense** :

CarDriver $\equiv$ *hasDrinvingLicense* `some` ***CarDrivingLicense***
and we declare the following individuals :

Driver*(Joe); hasAge(Joe, 20) ; hasName(Joe, "Joe")*
Person*(Toto); hasAge(Toto, 8) ; hasName(Toto, "Toto")*
Driver*(John); hasAge(John, 25) ; hasName(John, "John")*

then we author the following rule (i.e.car driving license)

car driving license :
`IF` *the age of the driver is more than 18*
`THEN` *add the car driving license to the driving licenses of the driver ;*

After the execution of this rule, we see in the ontology that for each driver who is more 18 (i.e. John and Joe), a car driving license is attributed. Then, we execute the car driver rule, that lists the names of all the car driver in the ontology. The result is : Joe is a car driver and John is a car driver because of the reclassification of Joe and John after executing the car driving license rule.

car driver :
`THEN` *print the name of the car driver + " is a car driver" ;*

Example 2 : In the ontology, we also define a concept ***ChildCarDriver****, subclass of a concept* ***Child****. A* ***ChildCarDriver*** *is a* ***Child*** *who has as father a*

CarDriver. *Then we define Toto who has father John.*

ChildCarDriver `subclassOf` ***Child*** `subclassOf` ***Person***
ChildCarDiver ≡ *hasFather* `only` ***CarDriver***
hasFather(Toto, John)

after executing the car driving license rule, we execute the child car driver rule that lists the names of all the child car driver. The result is that Toto is a child car driver because that in the ontology we define a child car driver as a person who has a car driver as father.

child car driver :
`THEN` *print the name of the child car driver + " is a child car driver" ;*

In the ontology, we also define the concept ***Contravention*** *that has an amount, a driver could have 0 or more contraventions and a contravention amount to pay.*

Contravention*(c1),* ***Contravention****(c2)*
hasContravention (John, c1), hasContravention (John, c2)

Using this definition, we author the remove car license rule that removes the car driving license for each car driver who has a contravention and calculates the contravention amount to pay. After executing this rule, John will loose his car license. Thus, when we re-execute the car driver rule and the child car driver rule, we will only have Joe is a car driver.

remove car license :
`definitions`
set 'a contravention' to a contravention ;
set 'a car driver' to a car driver where the contraventions of this car driver contain a contravention;
`IF` *'a car driver' is not null*
`THEN` *remove the car driving license from the driving licenses of 'a car driver';*
for each contravention in the contraventions of 'a car driver' :
- set the contravention amount to pay of 'a car driver' to the contravention amount to pay of 'a car driver' + the contravention amount of this contravention ;

In this example, the rule engine uses this inferred knowledge to trigger a computation that could not be easily represented in an ontology

Example 3 In the ontology, we define the concept ***Contravention*** *and the concept* ***RiskyDriver*** *as a* ***Driver*** *who has more than 3* ***Contravention*** *(a cardinality restriction), Frank as a* ***RiskyDriver*** *and we give to John 4* ***Contraventions****.*
RiskyDiver ≡ *hasContravention* `min` *3*
RiskyDriver*(Frank) ; hasName(Frank, "Frank")*

hasContravention (John, c1), hasContravention (John, c2), hasContravention (John, c2), hasContravention (John, c4) such as ***Contravention****(c1),* ***Contravention****(c2),* ***Contravention****(c3).* ***Contravention****(c4).*

If we execute the risky driver rule that lists the name of the risky driver we will only see that Frank is a risky driver which means that the classification engine is not able to classify John, who has four contraventions, as a risky driver.

risky driver :
`THEN` *print the name of the risky driver + " is a risky driver" ;*

2.2 Change-Management plug-in

The change-Management plug-in is for WODM enables to analyse the impact of ontology evolutions on business business rules. Ontology evolutions consist of changes that impact an ontology. These changes may be structural changes, conceptual changes, entity definition changes,... Business rules depends on the entities of the ontology and its evolution has an impact on the rule set that may causes inconsistencies. Thus, we developed the $\mathcal{MDR}$ approach ($\mathcal{M}$odel, $\mathcal{D}$etect, $\mathcal{R}$epair), which ensures the consistency maintenance of business rules while ontology evolution [3].

The $\mathcal{MDR}$ approach is based on design patterns and especially Change Management Patterns (CMP). This approach has been inspired from ONTO-EVO^{A}L [2], which deals with the consistency maintenance of OWL ontologies while they evolve. In our approach the CMPs are proposed to guide the evolution process of a rule set while maintaining its consistency. They consist of three categories of patterns :

1. *Change Pattern : used to model the ontology change knowledge that are important to detect its impact;*
2. *Inconsistency Pattern : used to detect the inconsistencies caused by a change;*
3. *Repair Pattern : used to propose solutions, called repair, to resolve the inconsistencies.*

The consistency maintenance process that we propose in our approach consists of three steps :

1. *Model the change to apply to the ontology using the change pattern;*
2. *Detect the eventual inconsistency that could be caused using the inconsistency pattern;*
3. *Propose repair to solve the inconsistency using the repair pattern.*

Change pattern is designed using an OWL ontology, called MDROntology, which model the ontology changes and their description, the inconsistencies that impact a rule set and their repairs. Each change has constraints to verify to avoid inconsistencies. Depending on the violated constraint the inconsistencies are detected using the inconsistency pattern. Thus, the inconsistency patterns are designed using a set of rules, called `Inconsistency Detection Rules` *(IDR),*

which in their condition part define the constraint that each change should verify and in their action parts define the inconsistency that will be caused by the change. The repair patterns are also designed using rules, called `Repair Rules` *(RR), which in their condition parts test on the detected inconsistency and on the modelled change then, in their action parts assign the repair(s) to apply. The rules designing the inconsistency and repair patterns have been authored over the MDROntology using the OWL plug-in (see section 2.1).*

Figure 1 illustrates the $\mathcal{MDR}$ process. As input of our system, the user models the change description as MDROntology individuals. The change description consists of :

- *Change type : add, remove, modify;*
- *Change object : conceptual change (i.e. subclass change, add concept, remove property,. . .) or entity definition change (enumeration change, restriction change, rename entity. . .);*
- *Change entities : concept or property that will be impacted by the change;*
- *Impacted rules and the scope of the impact (i.e. the impacted rule part).*

The change type, the change object and the change entities must be provided manually by the user. The impacted rules and the impacted rule parts are detected automatically. Nevertheless, depending on the change to apply other in-

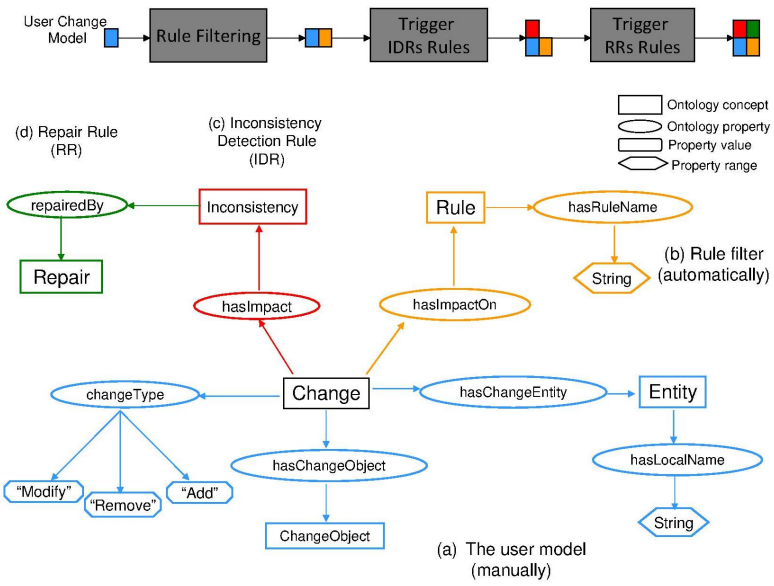

Fig. 1. $\mathcal{MDR}$ process

formation should be given. For example, the new collection of the enumeration

in case of an enumeration change, the new name of an entity in case of a rename entity change or the new range of a property in case of a range change.... In the following, the general template of a change pattern (see Fig. 1).

When the user launch the consistency maintenance plug-in within WODM, the modelled changes (MDROntology individuals) are loaded into the working memory and shown to the user through a user interface. When the user select one or more changes to apply, the IDRs are fired and detect the inconsistencies that will be caused. A general template of the inconsistency pattern is given below :

```
IF change.changeObject = changeObject
&& change.type = changeType
&& changeConstraint.satisfied = false
THEN change.inconsistency = inconsistency;
```

After the inconsistency detection and depending on the change to apply, the RRs are fored and one or more repair are proposed to the user, who will choose the repair to apply. The chosen repair will be automatically applied after verifying that it will not causes other inconsistencies. A template of the repair pattern is given below :

```
IF change.changeObject = changeObject
&& change.inconsistency = inconsistency
THEN inconsistency.repair = inconsistencyRepair;
```

3 Discussion

In section 1, we introduced the challenges we faced. In this section we discuss the challenges we resolved and those that we are trying to resolve.

To enable business users to deal with their business knowledge formalized using OWL Ontologies, we proposed an approach that consists of importing OWL ontologies into WODM. This approach enables authoring, in a natural controlled language, and executing rules over ontologies. Thus, business users are able to use the domain entities defined in the ontology to define business decisions using rules.

To import OWL ontologies into WODM, we performed an OWL to BOM mapping. Thus, when the users import an OWL ontology into WODM, the BOM is automatically generated and all the functionalities offered by the BRMS can be used.

WODM, or more specifically the BOM, is an Object Model. We cannot import all the expressiveness provided by OWL into such a model. Some constructs, such as `rdfs:subClassOf`, `owl:allValuesFrom`, `owl:inverseOf`*... are mapped. Some others, such as* `owl:someValuesFrom`, `owl: SymmetricProperty`, `owl:TransitiveProperty` *cannot be mapped into the BOM but they are processed at runtime (see Example 2 in section 2.1). Other constructs are neither mapped into the BOM nor processed at runtime such as* `owl:minCardinality`

(see Example 3 in section 2.1), `owl:maxCardinality`, `owl:complementOf`*... A complete description of the mapping can be found in [1].*

Ontologies evolve during their life cycle. Rules are authored over the ontology entities and depend on them; this is why an ontology evolution may make the rule set inconsistent. To make the rule set evolve with respect to the ontology while maintaining its consistency, we developed the $\mathcal{MDR}$ *approach, which is a pattern based approach. The general idea of this approach is that the user models the ontology change he wants to apply using the change pattern. Then, using the inconsistency patterns, inconsistencies that may be caused by the change are detected automatically. Finally, repairs that resolve the inconsistencies are proposed automatically thanks to the repair Pattern. Nevertheless, in the actual state of the work, the inconsistency patterns detect only two types of inconsistencies from six. A definition of business rules inconsistencies is done in [3].*

There are other challenges to be taken up; how to bring all the power of expressiveness of OWL to business users without loosing information? In the $\mathcal{MDR}$ *approach the inconsistency and repair patterns are defined manually which is a costly and not an easy task. Is it possible to automatically generate these patterns depending on the change to apply in a way that we will be able to detect all the inconsistencies that could impact business rules.*

References

1. *A. Chniti, S. Dehors, P. Albert, and J. Charlet. Authoring business rules grounded in owl ontologies. In M. Dean et al. (Eds.), editor,* RuleML 2010 : The 4th International Web Rule Symposium: Research Based and Industry Focused. *LNCS 6403, Springer-Verlag Berlin Heidelberg 2010, 2010.*
2. *R. Djedidi and M.A. Aufaure. ONTO-EVO^{A}L an ontology evolution approach guided by pattern modelling and quality evaluation.* Proceedings of the Sixth International Symposium on Foundations of Information and Knowledge Systems (FoIKS 2010)*, 2010.*
3. *M. Fink, A. El Ghali, A. Chniti, R. Korf, A. Schwichtenberg, F. Lévy, J. Pührer, and T. Eiter. D2.6 consistency maintenance. final report.* ONTORULE Delivrable, http://ontorule-project.eu/deliverables.*, 2011.*