# COMPARATION OF OWL ONTOLOGIES REASONERS
## *Testing Cases with Pellet and Jena*

José R. Hilera, Luis Fernández-Sanz and Adela Díez
*Department of Computer Science, University of Alcalá, Alcalá de Henares, Spain*

Keywords: Ontology, OWL reasoner, Jena, Pellet.

Abstract: This article describes the results of an evaluation process of four OWL reasoners: Pellet and other three included in Jena framework (OWL Default, OWL Mini, and OWL Micro). A simple ontology about programming languages has been used for the validation process carried out by reasoners and a Java program has been developed for testing different cases: reasoning over the original ontology model and testing different possible inference situations.

## 1 INTRODUCTION

One of the reasons for building applications based on ontologies comes from the fact that a reasoner can be used to infer additional assertions about the knowledge being modeled. One of the most popular frameworks for programming in Java applications using ontologies is Jena, which includes support for various types of reasoners through its API for inference (*jena.sourceforge.net*). A common feature of Jena reasoners is that they create a new model containing RDF triples resulting from the process of reasoning, but maintaining the set of original triples of the base model (Reynolds, 2010). When ontology is processed using Jena, it is possible to use a reasoner for inference. This article describes a work about the evaluation of four reasoners when new facts from the same OWL ontology are inferred.

## 2 EVALUATION

**OWL Reasoners to be Evaluated.** The Jena inference system was designed to allow different types of reasoners extract new knowledge. Normally, access to inference is achieved using the *ModelFactory* Java interface: this associates a dataset with a reasoner to create a new ontology model. This new model is composed of the assertions which were present in the original data, but also adding those ones derived from such data by applying rules or other inference mechanisms implemented by the reasoner. The Jena OWL reasoners could be described as instance-based reasoners, i.e., they use rules to propagate the "if and only if" implications of the OWL constructs on data instances. This approach contrasts with more sophisticated Description Logic reasoners which work with class expressions: they can be less efficient when handling instance data but more efficient with complex class expressions as well as able to provide complete reasoning. In this work an ontology represented in OWL is evaluated, using four reasoners: Pellet (2011) and other three reasoners which are included in Jena (OWL Default, OWL Mini, and OWL Micro). We define test cases and then compare the execution times and the results created by the different reasoners.

**Ontology for Test Cases.** We have used a simple ontology about programming languages to compare performance of the four reasoners. The formal knowledge modeled reflects some of the various categories used to classify programming languages depending on: the level of abstraction, the purpose, the historical development, and the programming paradigm. Figure 1 shows the classes included in the ontology. The main class is *ProgrammingLanguage*. Listing 1 presents the code about an instantiation of the class *ProgrammingLanguage*, in this case, the "*Java"* language. It can be noted that the property *belongsParadigm* has the value "*object oriented*", so it satisfies the restriction *equivalentClass* defined in the class *ObjectOrientedLanguage*.

**Software used for Evaluation.** The evaluation was done by modifying the ontology in order to test

R. Hilera J., Fernández-Sanz L. and Díez A..
COMPARATION OF OWL ONTOLOGIES REASONERS - Testing Cases with Pellet and Jena.
DOI: 10.5220/0003471404190422
In *Proceedings of the 6th International Conference on Software and Database Technologies* (ICSOFT-2011), pages 419-422
ISBN: 978-989-8425-77-5

different possible inference situations using the four reasoners. Listing 2 shows part of the Java code developed for validation, using the Jena API. An object using the *InfModel* interface must be created to test each reasoner previously. *InfModel* is an extension to the normal *Model* interface in Jena. It supports access to any underlying inference capability. After creating the object, the function *validate ()* is executed returning a validation report about the results of the reasoning. The time devoted to validation is measured.

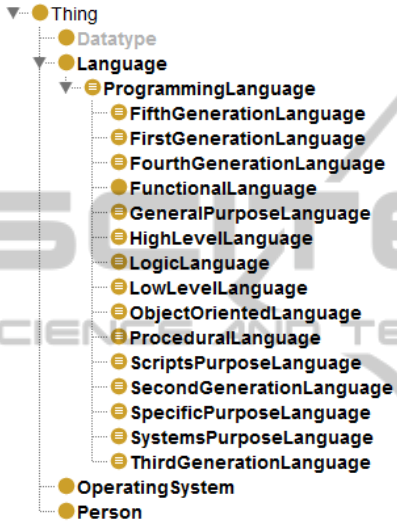


Figure 1: Classes in the ontology of programming languages.

```
<owl:Class
rdf:ID="ObjectOrientedLanguage">
 <owl:disjointWith>
  <owl:Class rdf:ID="FunctionalLanguage"/>
 </owl:disjointWith>
 <owl:disjointWith>
  <owl:Class rdf:ID="LogicLanguage"/>
 </owl:disjointWith>
 <owl:disjointWith>
  <owl:Class rdf:ID="ProceduralLanguage"/>
 </owl:disjointWith>
 <owl:equivalentClass>
  <owl:Restriction>
   <owl:hasValue>object oriented
   </owl:hasValue>
   <owl:onProperty>
    <owl:FunctionalProperty
     rdf:ID="belongsParadigm"/>
   </owl:onProperty>
  </owl:Restriction>
 </owl:equivalentClass>
 <rdfs:subClassOf>
  <owl:Class
rdf:ID="ProgrammingLanguage"/>
 </rdfs:subClassOf>
</owl:Class>
```

Listing 1: Ontology (class "ObjectOrientedLanguage").

```
<ProgrammingLanguage rdf:ID="Java">
 <belongsGeneration>third
 </belongsGeneration>
 <belongsParadigm>object oriented
 </belongsParadigm>
 <hasExtension>.class</hasExtension>
 <hasExtension>.jar</hasExtension>
 <hasExtension>.java</hasExtension>
 <hasLevel>high</hasLevel>
 <hasReservedWord>void</hasReservedWord>
 <hasReservedWord>public</hasReservedWord>
 <isCreatedBy>
  <Person rdf:ID="JamesGosling"/>
 </isCreatedBy>
 <isCreatedIn
  rdf:datatype="&xsd;gYear">1995
 </isCreatedIn>
 <supportsOperatingSystem>
  <OperatingSystem rdf:ID="MacOSX"/>
 </supportsOperatingSystem>
 <supportsOperatingSystem>
  <OperatingSystem rdf:ID="Linux"/>
 </supportsOperatingSystem>
 <supportsOperatingSystem>
  <OperatingSystem rdf:ID="Windows"/>
 </supportsOperatingSystem>
 <supportsOperatingSystem>
  <OperatingSystem rdf:ID="Solaris"/>
 </supportsOperatingSystem>
</ProgrammingLanguage>
```

Listing 2: Ontology source code (individual "*Java*").

```
//Validation with OWL Default reasoner
long startTime= System.currentTimeMillis();
InfModel im = ModelFactory.createInfModel
  (ReasonerRegistry.getOWLReasoner(),
   modelOWL.getRawModel());
ValidityReport vr = im.validate();
if (vr.isValid())
  System.out.println("Correct model");
else
 for(Iterator<Report>I = vr.getReports();
  i.hasNext();)
  System.out.println("____+++___"+i.next());
long endTime = System.currentTimeMillis();
System.out.println(endTime-startTime);
```

Listing 3: Java code for evaluation of time for reasoning.

**Test Case 1: Validation of the Original Model**. In this first case we used the original ontology, only considering cardinality constraints and values applied to the parent class and to child classes generated by the value of the functional properties that define a particular classification. As seen in Table 1, the results for all the reasoners are the same, resulting in a correct validation of the model, but with significant differences in the execution time. The time taken by the default reasoner is four times the one by OWL Mini, because this avoids infinite expansions when *bNodes* are included, where restrictions as *minCardinality* or *someValuesFrom* enter in the process, so it leads to a significant performance improvement. Validation with OWL Micro is similar in time values to the ones by Pellet, with an execution time 56 times

lower than the default reasoned: this is possible because it restricts its functionality to RDFS hierarchies, and *intersectionOf*, *unionOf* and *HasValue* axioms.

Table 1: Results of the test cases for validation of restrictions (execution time in "ms" and error messages).

| Test case | OWL Default | OWL Mini | OWL Micro | Pellet |
|---|---|---|---|---|
| 1 | 21036 | 7635 | 855 | 1022 |
| 2 | 49902 | 6440 | 505 | 727 (KB is inconsistent) |
| 3 | 38071 Too many values, Conflict | 12342 Too many values, Conflict | 1073 Conflict | 977 KB is inconsistent |
| 4 | 71494 | 13912 | 1029 | 1080 |
| 5.1 | 187200 Too many values | 18056 Too many values | 1392 Too many values | 1144 KB is inconsistent |
| 5.2 | 48068 | 10786 | 1002 | 1136 |
| 5.3 (val.2) | 52742 | 14490 | 1142 | 1096 (KB is inconsistent) |
| 5.3 (val.1) | ---- Code exception | 27276 Too many values | 1235 | 1152 KB is inconsistent) |

**Test Case 2: Validating Value restrictions on Data.** This test case requires changing the original model by altering the values of the properties that settle ratings for any of the defined individuals. For example, the value of the property *belongsParadigm* has been changed in the case of the individual "*Java*", from "*object oriented*" to "imperative". As a value not covered by the range of data that has been assigned to the property domain, the validation of the resulting model would be incorrect. Table 1 shows the results. While all reasoners incorporated in Jena get a successful validation, Pellet recognizes the inconsistency in the model and launched a bug report. The cause might be found in the documentation on Jena inference: "The critical constructs which go beyond OWL lite and are not supported in the Jena OWL reasoner are *oneOf* and *complementOf*" (Reynolds, 2010). Due to the transformation of *DataRange* as combined lists with *oneOf* in the modified ontology, the reasoners skip checking this type of construction and continue validating the rest of the model, returning a positive result.

**Test Case 3: Validating Cardinality Restrictions on Functional Properties.** A functional property is one that can take only a single value for each instance. We have added in the ontology a property *belongsGeneration* in the case of individual "*Prolog*" and it is set to "*fourth*". Since this kind of special properties are not sensitive to context (in a global level), an error should emerge in the validation when a second literal is added. In this case, all reasoners notify errors, but with different error reports (table 1). The differences between OWL Mini and Default are minimal. Most notable differences appear in the case of OWL Micro reasoner, which suppress the validation of the functional property value, but recognizes that there is a case of incompatibility between disjoint classes. The validation would have been positive if it had been declared as a value of this property, an out of range value, because this checking is not implemented in this reasoner, so it would have not provoked inconsistency between classes (there would be no disjunction between classes). The error launched by Pellet just affects the value of the functional properties, rather than class operations.

**Test Case 4: Validating *minCardinality* Restrictions.** In this test case, we have defined a lower value for the same type properties to the same instance, so that they remain below the declared value for that restriction. It has been eliminated the property *isCreatedBy* of the individual "*SQL*" so reference to the creators of this language has been missed. With this scenario, a priori, the resulting model validation should fail. However, as shown in table 1, the validation with all the four reasoners is correct, showing an unexpected result. This happens because, as in other ontology languages, the semantics of OWL adopt the Open World Assumption (OWA) approach. OWA model holds that an agent or observer cannot have a complete view of the world, and therefore it cannot make assertions about facts which are unknown. This involves, directly, an incomplete inference process with open world reasoners, against those based on closed world assumption, in which the lack of information is automatically translated into an assertion of falsehood. In essence, we can say that it is wrong to expect that the absence of information in the OWL model validation will generate an error, taking into account the principles governing OWA, and adopted by RDF and OWL. For a simulation of the closed world assumption, it is possible to use the Jena framework "Eyeball" and the new integrity checker that offers the Pellet reasoner for OWL.

**Test Case 5: Validating *maxCardinality* Restrictions.** This case requires reversing the changes to the previous point, i.e., we defined a larger number of the same type of properties on the same instance, so that they are higher than the

declared maximun value for that restriction. There are no problems in the case of working with *DataType* properties, but in the case of *Object* type properties, we can test the differences when individuals are different or not. We have validated the following possibilities:

1) **Exceeding the Maximum Cardinality of a DataType Property.** We changed the restriction associated to the property *hasExtension*, defining as *maxCardinality* the value "2". With this assumption, should fail in the case of the individual "*Java*" of the class *lenguajeProgramacion*, which has three properties of this type. In this case, the reasoners generate incorrect validation reports (table 1). The Default OWL and OWL reasoners behave in the same way. There are two types of warning in reports: one about the classes and subclasses to which the individual "*Java*" belongs and another one referred to nodes with names related to the ID assigned to each constraint that define each subclass. In the case of OWL Micro reasoner the message only shows the first occurrence where there is conflict and stops validation. Pellet also shows a validation message that is enough to verify that it has exceeded the value of a particular property, but it does not show what the individual or class is affected.
2) **Exceeding the Maximum Cardinality of an Object Property without defining the Individuals involved as Different.** We changed one of the restrictions, *supportsOperatingSystem,* defining a maximum cardinality "2", in order to see how it acts on individuals (as "*Prolog*", "*SQL*", "*Java*") which have assigned more than 2 operating systems. We can realize in this case the validation is successful with all the reasoners (table 1), because we have to explicitly state that individuals are different for the cardinality constraints operate as desired although each of the three individuals has various instances of *supportsOperatingSystem* property. Here it is a consequence of the paradigm OWA because it cannot cause any definitive deductions whether knowing if the individuals are identical or not.
3) **Exceeding the Maximum Cardinality of an Object Property identifying the Individuals involved as Different.** We took the same case as above excepting that define the operating systems as different from each other (for this example, four individuals: "*Windows*", "*Linux*", "*MacOSX*" and "*Solaris*"). To do this, the individual "*SQL*" of *ProgrammingLanguage* was modified by adding the four instances for the property *supportsOperatingSystem*. In this case, validation should be wrong for all reasoners. But Pellet reasoner was the only one that detected that cardinality is exceeded for any individual. This is because the sublanguage on which are built the OWL Jena reasoners, OWL Lite, only supports 0 or 1 as cardinality constraints (Reynolds, 2010). So we can consider this limitation and change the value of the maximum cardinality from 2 to 1. The validation with this change produces the results shown in table 1. In this case, the OWL default reasoner gets hooked in an infinite loop trying to insert blank nodes for all generated classes that take the cardinality constraint, throwing an exception message. OWL Mini reasoner is very helpful to avoid this because it prevents these expansions and performs validation controlling this inconsistency in the model. Finally, Pellet also locates directly this inconsistency.



## 3 CONCLUSIONS

From results of test cases we can conclude that the reasoners embedded in Jena to provide inference and validation are incomplete and with important limitations. In the case of OWL reasoners, the validation capacity is quite limited, assuming as valid cardinality restrictions broken in the ontology. In these situations, the external reasoner Pellet provides a more complete reasoning based on OWL DL version with shorter response times. Moreover, certain aspects of the programming interface are obsolete, as Jena has been built based on OWL 1.

## ACKNOWLEDGEMENTS

This research has been partially funded in Spain by CAM and UAH (grant CCG10-UAH/TIC-5915).

## REFERENCES

Pellet, 2011. *Pellet: OWL 2 Reasoner for Java*, Clark & Parsia. http://clarkparsia.com/pellet.

Reynolds, D., 2010. *Jena 2 Inference support*, souceforge. http://jena.sourceforge.net/inference/index.html.