CBR and Ontology Code Guide

Hugo Muñoz Hernández

Project developed at ISAE-SUPAERO, 10 Avenue Edouard Belin, 31055 Toulouse.Department of Complex Systems Engineering.

Supervisors: Juan José MONTERO JIMÉNEZ

Rob VINGERHOEDS

Contents

1 Installation and configuration

For the installation procedure described here it is assumed that the user has installed a version of *Eclipse* that supports at least *Java* 15. Under the previous condition, the following steps may be followed to ensure the correct installation:

1. Download the *.zip* compressed folder of the project and save it in a selected local address in your computer.

2. Make right click over the compressed folder and use *Extract Here*, a decompressed folder with the same denomination has been created. Inside this folder, another folder named *InternshipProject* may be found.

3. Open *Eclipse* and browse to select *InternshipProject* as the working folder.

4. Once *Eclipse* has been initialized, go to *File* → *Import* → *General* → *Existing Projects into Workspace* → *Browse* and select the folder *InternshipProject.* The project will be built in the current workspace of *Eclipse.* The recognition of the folder as an *Eclipse* project is possible because of the file *.project* in the same folder.

5. In the case that some referenced libraries (*.jar*) seem to be missed, right click on *InternshipProject* at the workspace menu on the left → *Build Path* → *Configure Build Path* → *Libraries* → click on *Classpath* in the list below and now the option buttons on the right are available. Delete the paths that are indicated as erroneus and click *Add JARs* on the left to reintroduce the libraries that are missed. Browse to the *external-libs* folder adn select the *.jar* files that have to be restored, then just click OK and *Apply and Close.* However, this problem is not likely to occur as the *.classpath* file store the path to all the dependencies of the project. Another possible solution to the problem if this existed would be to open the *.classpath* file with a plain text editor and change the paths that are referencing a local folder in another user computer to a general path starting at the folder *external-libs* of the project.

Now that the code is installed, some configurations are needed to star execution. There is a class named *AppConfiguration* in the *User* package which only contains public static attributes and no methods. These attributes are read and used by other classes of the project, so they are stored together in the accessory class *AppConfiguration* to be accessed (not modified) by other pieces of code in the application. The very first change that the user must make in the mentioned class before executing the code is setting the *data˙path* attribute to the actual local path of the *data* folder of the project. From that point, the attributes on *AppConfiguration* should be updated to the file names that are wanted to be used inside the *data* folder (*.csv, .owl,.prj*, etc). See the *javadoc* of the project and the comments in the source code of *AppConfiguration* to get known about the meaning of the attributes.

2 Basic usage

The usage of this application will be performed simply by the execution of java classes from Eclipse (or any other IDE). It may also require the manipulation of files (ontology files, *.csv*, *myCBR* project files, etc) in the designated file folders. Unless a modification of the source code is needed for some reason, there are 5 executable classes in the project that concern to the user at this moment: *CSVtoOntologyExec*, *myCBRSetting*, *SPARQL*, *GUI2* and *GUI3*.

2.1 Load data in a table format (*.csv*) to an ontology file (*.owl*) using *CSVtoOntologyExec*

When executed, the class *CSVtoOntologyExec* will read the content in the specified data base contained in a *.csv* file and load the information into an ontology file (*.owl*). An important consideration is that the *.owl* files should be written in *RDF/XML* syntax (choose that option when using *Save as* in *Protégé*).In what concerns to *myCBR* retrieval, this class may only be executed when the case base information and similarity values contained in the *.prj* file should be changed or updated to perform CBR queries using new data.The path for working file folder should have been set in the *AppConfiguration* class, together with the corresponding files denominations. In this folder, it is recommended to locate a clean version of the ontology that is wanted to be used (without instances or individual or property assertions) and another copy (of course with a different name) where the data base will be stored. So, when updating the data base the procedure should be as follows:

- Delete the data base ontology file.
- Make a copy of the clean ontology file and change the name as desired . Of course, the same name should be specified in the *AppConfiguration* class (*ont_file_name*).
- Execute *CSVtoOntologyExec*, which will read the clean ontology file (specified in *AppConfiguration* as *base_ont_file_name*) and the data base of the *.csv* file (specified in *AppConfiguration* as *csv*) to merge the information in the ontology data base file.

The translation of the information stored in the table to ontological entities is stated with the appropriate using of the methods of the class *CSVtoOntology* in the package *OntologyTools*. See the *javadoc* of the project for details. By this way, the code is flexible to adapt to the ontological meaning of the content in the different columns and cells forming the table tabular data base.

In this particular case, the executable code in the *CSVtoOntologyExec* class is configured for the Predictive Maintenance data base and the *OPMAD* ontology. Nevertheless, the class could be modified if needed to suit to another different case or to adapt to a restructuring of the current data base and ontology.

2.2 Preparing project case base and similarity values for *myCBR* with *myCBRSetting*

The class *myCBRSetting* may be executed when the data base in the *.csv* (file name specified as *csv* in the class *AppConfiguration*) file or the ontology *.owl* file have been changed (normally their modifications are coordinated). In order to perform *myCBR* queries on a new case base, or a case base that has been updated, the information must be written in the *.prj* (file name specified as *projectNmae* in the class *AppConfiguration*) file, which is the one that contains the project data for *myCBR*. Moreover, the class will run an *OWL* reasoner (more precisely the *HermiT* reasoner, see Section 3) on the ontology that will check its consistency and infer relations for the knowledge contained in the ontology. In consequence, the execution of the class can take a bit of minutes, depending on the ontology size.The main reason to use the reasoner is to be able to perform queries on the ontology to extract information that can be used to calculate ontological similarity values. Moreover, the *HermiT* reasoner allows to make queries about relations that are not initially explicit in the ontology but inferred during the reasoning process. Obviously, if the ontology is not consistent the execution will stop, so to work with an ontological data base it must be consistent.

At first, the class uses an instance of *CBREngine* to load the current *.prj* file, delete all the existing instances in the case base and introduce the new ones, with their corresponding attribute values, by importing the chosen *.csv* file. Then, the similarity functions are established with the appropriate values. In particular, for the field corresponding to the Predictive Maintenance task associated to each one of the cases, the similarity values are calculated using an ontological method. The querying of the ontology uses the classes *DLQueryEngine* and *DLQueryParser* of the package *OnotlogyTools*. See the *javadoc* of the project for more details.

2.3 Query and retrieval using the *GUI*'s

Once executed *CSVtoOntologyExec* to load the data base into the ontology and *myCBRSetting* to configure the .prj file for *myCBR*, only the executable *GUI*'s are required to query and retrieve until the case base is wanted to be modified. The tool *myCBR* uses the case base, the attributes and the similarity functions specified in the *.prj* file to search the most suitable case for the given query. To visualize or to modify manually the project file the *myCBR Workbench* application may be used.

Case variable	Variable type (*myCBR*)	Values
Task	Symbol	Feature extraction, Fault detection, Fault identification, Health modelling, Health assessment, Remaining useful life estimation, One step future state forecast, Multiple steps future state forecast.
Case study type	Symbol	Rotary machines, Reciprocating machines, Electrical components, Structures, Energy cells and batteries, Production lines, Others.
Case study	String	A string will be typed for the query. The similarity is 0 if no equivalent string is found in the case base for this field. The Levenshtein function of *myCBR* allows to support slight mispelling.
Input type	Symbol	A list of variables must be provided separated by ', ' and where all the words should begin with capital letters as it is established in the case base. An additional Levenshtein method in *Java* allows to support mispelling up to 3 erroneus characters.
Online/Off-line	Symbol	Online, Off-line, Both, Unknown synchronzation.
Input for the model	Symbol	Signals, Structured text-based, Text based maintenance/ operations logs, Time series.
Publication year	Integer	This field is not provided by the user, the application will used the current date automatically. The most recent cases in the case base will be prioritized over the older ones.

Table 1: Case variables for querying and retrieval

When executing any one of the *GUI's*, the class *Recommender* is used. Most of the similarity functions are defined during the execution of the class *myCBRSetting*, but there is one particular field which similarity values are defined by comparing the current query to all the cases in the data base individually, and that is *Input type*. An analog method is used, which is analog to the one applied to establish the similarity values between the different Predictive Maintenance functions (field *Task*).

Two executable *Graphical User Interfaces* are provided for querying: *GUI2* and *GUI3*.

2.3.1 *GUI2*

The *GUI2* allows the user to perform one query at a time by specifying the following parameters:

- Values of the case fields for the retrieval. Some of them must be typed (*Case Study* and *Input Type*) and for the rest of the fields the values are selected form the ones available in a drop down menu.

- Value of the weights assigned to each field.
- Type of amalgamation function that is used to get the global similarity value of each case.
- Number of cases to be retrieved. The resulting list of cases (ordered from higher to lower similarity value) will be shown in the screen after submitting the query.

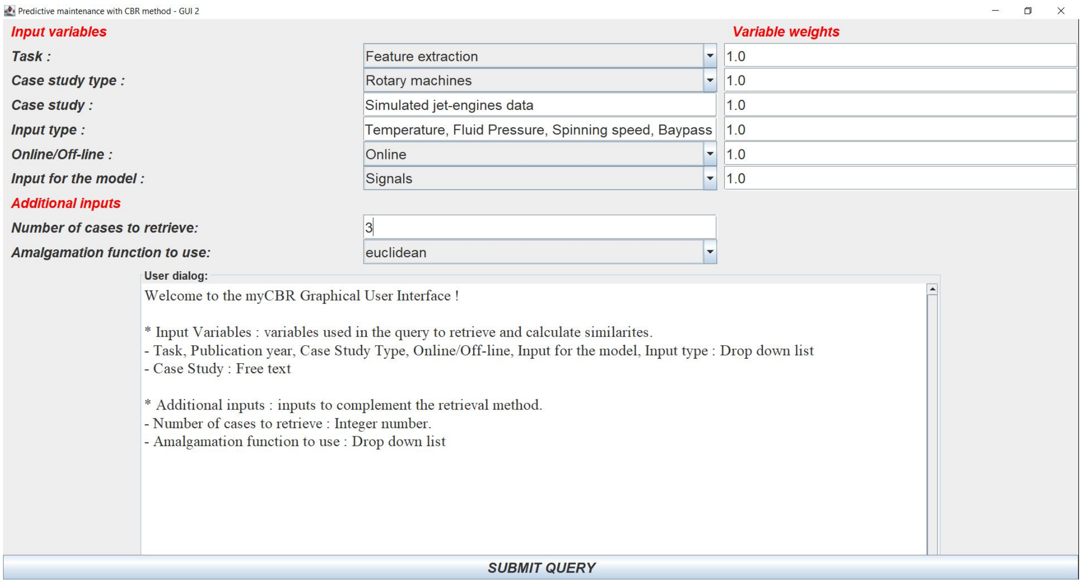

Figure 1: Window of the *GUI2*

If one of the fields is left blank, then its weight in the global similarity value is automatically set to 0. The value of the field *Case study* is expected to be equal to one already existing in the data base, otherwise the similarity value will be 0 for all the cases. The field *Input type* contains a list of variables (most of them physical variables) that are considered in the Predictive Maintenance model of each case. This list must be typed with the terms separated by ', ' and all the words starting by capital letters, as they appear in the case base. Nevertheless, both fields can manage with possible misspelling errors using the Levenshtein distance method. In particular, the *Case study* field is defined as string type in *myCBR*, and it uses the default Levenshtein comparison function. But, for the *Input type* field, the Levenshtein method is not available in *myCBR* as it is declared as symbolic value. So, an additional method (class *LevenshteinDistanceDP*) to allow up to a distance of three erroneous characters in the spelling is implemented in the similarity function definition (*Recommender*). See the *javadoc* for more details.

2.3.2 *GUI3*

Using this *GUI*, the user is able to execute a list of consecutive queries provided in an input file in *.csv* format and to save their results in separate files (one for each query in the list). It is necessary to prepare an input file with the appropriate structure (see input˙file), like the one shown in input˙file. An example of input file is also provided in the actual project folder of the application. For the fields that are stated as symbolic in *myCBR* and for the string variable *Case study*, the user must type in the input file a value which is included among the possible values for each field (see Table 1), otherwise the similarity will be just 0. Symbolic fields, with the exception of *Input type*, do not support misspelling. The field *Input type* contains a list of variables separated by ', ' where the words should begin by capital letters (as they are in the data base). As soon as one of the variables of the list exist in one of the cases in the data base the similarity value will not be 0 for that case.

Task	w1	Case study ty	w2	Case study	w3	Online/Offline	w4	Input for the model	w5	Input type	w6	Number of ca	Amalgamation function
Fault detectio	1	Rotary machi	1	Simulated jet-	1	Online	1	Time series	1	Temperature	1	20	euclidean
Feature extra	1	Rotary machi	1	Simulated jet-	1	Offline	1	Time series	1	Temperature	1	1	euclidean
Fault detectio	1	Rotary machi	1	Simulated jet-	1	Offline	1	Time series	1	Temperature	1	5	euclidean

Figure 2: Example *.csv* input file for the *GUI3*

After having prepared the input file, the *GUI* window will just require to the user to provide the name of the input file and also that of the result files, as shown in Figure GUI3image. For each one of the queries in the list, a result file will be generated with the denomination specified by the user and an index added to the name indicating its position in the list of queries. An example result file is shown in the Figure 4.The content of the result files is another list containing the information about the cases that have been retrieved (as many as demanded by each query).

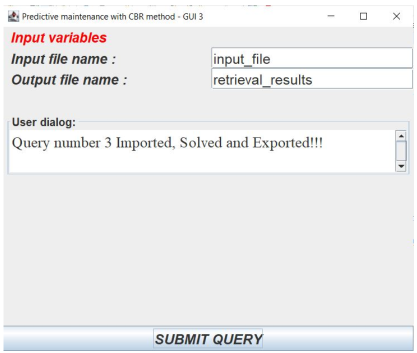

Figure 3: Window of the *GUI3*

Reference	Sim	Task	Case study typ	Case study	Online/Off-lin	Input for the r	Model Approa	Models	Input type	Number of inp	Performance	Performance	Complementa	Publication identifier
209	0,919	Fault detectio	Rotary machi	Simulated jet-	Online	Time series	Single model	Voting metho	Spinning spee	3	N/A,	N/A,	No info about	doi: 10.12700/APH.15.1.2018.2.10
191	0,901	Fault detectio	Rotary machi	Simulated jet-	Online	Time series	Multi model	Bayes model,	Measurement	2	Robustness , I	N/A,	10 operationa	doi :10.1109/TCST.2011.2177981
166	0,845	Fault detectio	Rotary machi	Wind turbines	Off-line	Time series	Single model	Gaussian proc	Temperature	22	Accuracy,	<0.945,	one case of s	https://doi.org/10.1016/j.renene.2018.10.088
167	0,845	Fault detectio	Rotary machi	Wind turbines	Online	Time series	Single model	Gaussian proc	Wind power	22	N/A,	N/A,	one case of s	https://doi.org/10.1016/j.renene.2018.10.088
210	0,837	Fault identific	Rotary machi	Simulated jet-	Online	Time series	Single model	Expert system	Results of a fa	1	N/A,	N/A,	No info about	doi: 10.12700/APH.15.1.2018.2.10
211	0,818	Fault detectio	Rotary machi	Simulated jet-	Off-line	Time series	Multi model	Hybrid Kalmar	Compressor T	8	Visual indicat	N/A,	6 operation m	doi: 10.1109/ACC.2013.6580567
192	0,818	Fault identific	Rotary machi	Simulated jet-	Online	Time series	Multi model	Bayes model,	Measurement	2	Robustness , I	N/A,	10 operationa	doi :10.1109/TCST.2011.2177981
195	0,786	Fault detectio	Rotary machi	Rotor shafts	Online	Time series	Multi model	Gauss-Marko	Imbalance so	2	N/A,	N/A,	No info about	doi.org/10.1117/12.475502
72	0,772	Fault detectio	Structures	Tank reactor	Online	Time series	Multi model	Updated Rule	Temperature,	3	Probability,	From graphics	2 operational	doi:10.1016/j.ejor.2010.03.032
98	0,760	Fault detectio	Rotary machi	Steam Turbine	Online	Signals	Single model	Extreme Grad	Condenser va	7	N/A,	N/A,	No info about	doi.org/10.1016/j.microrel.2013.03.010
59	0,758	Remaining use	Rotary machi	Aircraft bearir	Online	Time series	Multi model	Relevance ve	Health index,	1	Score functio	19,66,	No info about	doi.org/10.1016/j.ress.2017.12.016
58	0,758	Remaining use	Rotary machi	Aircraft bearir	Online	Time series	Multi model	Ensemble lea	Health index,	1	Score functio	7,8,	No info about	doi.org/10.1016/j.ress.2017.12.016
60	0,758	Remaining use	Rotary machi	Aircraft bearir	Online	Time series	Single model	Particle filter,	Health index,	1	Score functio	301,8,	No info about	doi.org/10.1016/j.ress.2017.12.016
204	0,756	Remaining use	Rotary machi	Wind turbines	Online	Time series	Single model	Geolocation p	Distance, Deg	2	Prognostic ho	(65),(0.7),	No info about	doi.org/10.1016/j.renene.2017.05.020
2	0,755	Health assess	Rotary machi	Simulated jet-	Off-line	Time series	Single model	Logistic regre	Temperature,	5	Mean absolut	(<0.05),	1 operational	doi.org/10.1016/j.ast.2018.09.044
1	0,755	Health modell	Rotary machi	Simulated jet-	Off-line	Time series	Single model	Logistic regre	Temperature,	5	Mean absolut	(<0.05),	1 operational	doi.org/10.1016/j.ast.2018.09.044
51	0,754	Remaining use	Rotary machi	Simulated jet-	Off-line	Time series	Multi model	LSTM (Long-S	Temperature,	14	Root mean sq	6,9±4,7,	6 CMPASS dat	doi.org/10.1016/j.ast.2019.105423
53	0,754	Remaining use	Rotary machi	Simulated jet-	Off-line	Time series	Single model	Recurrent Ne	Temperature,	14	Root mean sq	10,2±5,8,	6 CMPASS dat	doi.org/10.1016/j.ast.2019.105423
52	0,754	Remaining use	Rotary machi	Simulated jet-	Off-line	Time series	Single model	LSTM (Long-S	Temperature,	14	Root mean sq	8,2±5,7,	6 CMPASS dat	doi.org/10.1016/j.ast.2019.105423
54	0,754	Remaining use	Rotary machi	Simulated jet-	Off-line	Time series	Single model	Gated recurre	Temperature,	14	Root mean sq	10,0±6,0,	6 CMPASS dat	doi.org/10.1016/j.ast.2019.105423

Figure 4: Example *.csv* result file for the *GUI3*

2.4 *SPARQL* queries

A simple executable class has been added to the project to execute *SPARQL* queries on the working ontology in the same way that they are available in *Protégé*. The *Jena* tool (see dependencies for more information) is used for that purpose.The *SPARQL* queries allows the user to obtain an accurate required information from the ontology. A general reference for the *SPARQL* language can be found at [11].The user just needs to write the query in a correct syntactical form and execute the class file to get the results of the query on the console. These queries are very quick in their execution as they do not need to run a reasoner and check the ontology consistency. Even if the current implementation in the project is accessory, the *SPARQL* queries could be potentially used to extract information from an ontology for the definition of ontological similarity values.

2.5 *SQWRL* queries

The implementation of the *SWRL API* together with the reasoning engine *drools* allows to add *SWRL* rules to the working ontology execute *SQWRL* queries. See dependencies for more information. The *SQWRL* language, which is based on the *Semantic Web Rule Language* (*SWRL*), opens the possibility to accurate queries with a relatively simple syntax. There is a paper [21] where the creator of this language give an introduction to the main rules and basic syntax of the queries. An inconvenient has been found for the use the *SQWRL* queries for the application case in the current research project: the reasoner must be run once for each query, so, when working with ontologies having a big number of individuals declared, this process can require an important amount of *RAM* memory in the computer and it can last long if many queries are wanted to be executed. That is why, at this moment, the direct querying through the *OWL API* is used for extracting information from an ontology, as it only requires to run the reasoner (*HermiT*) once before executing as many queries as desired. Indeed, the example class in the project *SWRLAPIexec* allowing to execute *SQWRL* queries may not work correctly due to the compatibility problems of *SWRL API* with the latest version of *OWL API*. See dependencies. However, as it is potentially useful for the development of the project, the *SWRL API* implementation is still included in the project.

2.6 The *javadoc*

A complete *javadoc* has been generated for this project. It is available in the folder *javadoc* of the project. The easiest way to access to the documentation is opening the *index* file in the mentioned folder. It is an *HTML* file in the standard format for *javadoc* resources available in the web, and it may be opened with a web browser. The documentation contains a detailed descriptions of all the classes and methods organized in linked pages, allowing to navigate through the structure of the code. To update the documentation using *Eclipse*, go to *Project* → *Generate Javadoc*. When the window is open, the user must select the folder where the *javadoc* will be saved. It is recommended to use the folder *javadoc*, after having deleted the previous content to avoid problems. Of course, the generation of *javadoc* depends on the *javadoc* format comments that have been added to the code. See [12] for more details on the *javadoc* comments format.

2.7 Management of the *data* folder

The purpose of the *data* folder in the project is to store all the files that are needed for the execution tasks of the application. It is very important to remember that the denomination of the files that are wanted to be used inside this folder must be in coordination with the names introduced in *AppConfiguration*, so as the code is able to find such files. Some of the main types of files and their function are listed:

- Files *.owl* : the ontology files. One important consideration is that these files must be written in the *RDF/XML* syntax (choose that option when using *Save as* in *Protégé*). They can be opened with *Protégé* to visualize and edit the ontology or with a simple plain text editor, which allows to modify manually the statements. Two files may be used, one file should contain a clean ontology (without the declaration of the individuals in the data base) and the other one with all the data base loaded (and a different name, of course).
- Files *.csv* : files with a table format. A data base in *.csv* format may be needed to be loaded in the working ontology. The input and result files used by the *GUI3* are also *.csv* format. These files can be opened with *Excel*. Using a plane text editor to open *.csv* files may be recommended to ensure that no weird characters have been introduced by *Excel* at the beginning of the data (an unusual bug in *Excel* that may alter the name of the first column of data).

- Files *.prj* : the project files of *myCBR*, where all the information concerning the query and retrieval must be stored. This type of file can be opened with the *myCBR Workbench* application.
- Files *.ttl* : these files contain the ontology dependencies (BFO, CCO ontologies, etc). They are necessary to read the ontologies when working with local imports, so as the ontology is independent of the online servers. They are available at the GitHub repository [22], and may be updated with newer versions from time to time. For the execution of the code, some mappers are set using *OWL API* to link the *.ttl* files with the *URI* that are used by the ontologies to invoke the corresponding dependencies.
- File catalog-v001.xml : this file stores the paths to allow *Protégé* to import the local ontology dependencies in *.ttl* format when opening an ontology file that needs those dependencies.

3 Dependencies

Here are listed the main *APIs* and libraries which are needed for the application, which are included in the *external-libs* folder of the project:

- *OWL API* : it is an *API* that allows to read, modify, manipulate and create *.owl* ontologies. The version currently used in the application is 5.1.9, but future updates could be possible. The *API* is implemented by including the necessary *.jar* files in the *external-libs* folder of the application. The last version of *OWL API* is available at the *Maven* repository [2] (artifact owlapi-distribution), but it has been directly obtained as *.jar* files from [1] with all dependencies. Previous versions are also available. A complete documentation can be accessed at [3] and there is an introduction tutorial written by the creators at [6]. The licenses concerning the *API* are the Apache License [4] and the GNU license [5].

- *HermiT* reasoner: it is a reasoner that works in *Protégé*, the most used software for ontology manipulation. There is and implementation for *Java* based on the *OWL API*. The version used is the latest one (1.4.5.519), and it can be found at the *Maven* repository [7] or downloaded from [8] with all dependencies. The reasoner is needed to perform queries on ontologies through *Java*.The documentation for a previous equivalent version (1.3.8.4) is available at [9]. The GNU license es applicable [5].

- *myCBR*: it is an open source tool for case-based reasoning applications. The *Software Development Kit* of *myCBR* project has been implemented in the application with the appropriate *.jar* file. The *myCBR Workbench* application may be very useful for visualizing *.prj* files. All the information concerning *myCBR* project (source code, installation guide, tutorials, javdoc, etc) is available at the website [10].

- *Jena*: it is a tool for ontology manipulation. In paritcular, it is used in this application for adding the capability to perform *SPARQL* queries on ontologies through *Java* with the *SPARQL* executable class in the project, that uses the *Jena* methods. The latest version is used (4.0.0), and it can be found at the *Maven* repository [13] (artifact jena-arq) or downloaded from [14] with all dependencies. A complete documentation of the *Jena Core* is available at [15].The Apache license [4] is applicable.

- *SWRL API* with *drools* rule engine: the *SWRL API* may be used to implement *SWRL* rules in an ontology through a *Java* application. Moreover, if the *drools* reasoner is added, the application is able to execute *SQWRL* queries on the working ontology. In this case, the latest version of *SWRL API* (2.0.9) is used, which is available at the *Maven* repository [16]and at [17] for direct *.jar* download with all dependencies. A *javadoc* is available at [18]. Nevertheless, the compatibility of the *SWRL API* with *OWL API* is only guaranteed up to version 4.5.9, even if some later versions could be supported. Moreover, to query an ontology with the *SWRL API* and *SQWRL* language, an additional implementation of a reasoner is necessary: the implementation of *drools* engine is available for this purpose. Once again, it may be found at the *Maven* repository [19] or for direct *.jar* download with all dependencies at [20].

References

[1] JAR download (April 2021): https://jar-download.com/artifacts/net.sourceforge.owlapi/owlapi-distribution

[2] MVN repository (April 2021): https://mvnrepository.com/artifact/net.sourceforge.owlapi/owlapi-distribution/5.1.17

[3] OWL API javadoc (April 2021): https://javadoc.io/doc/net.sourceforge.owlapi/owlapi-distribution/latest/index.html

[4] Apache License version 2.0: https://www.apache.org/licenses/LICENSE-2.0

[5] GNU LESSER GENERAL PUBLIC LICENSE: http://www.gnu.org/licenses/lgpl-3.0.txt

[6] MATENTZOGLU Nicolas., PALMISANO Ignazio. *An introduction to OWL API.* University of Manchester, 2016.

http://syllabus.cs.manchester.ac.uk/pgt/2020/COMP62342/introduction-owl-api-msc.pdf

[7] MVN repository (April 2021): https://mvnrepository.com/artifact/net.sourceforge.owlapi/org.semantic

[8] JAR download (April 2021): https://jar-download.com/?search˙box=org.semanticweb.hermit

[9] HermiT 1.3.8.4 javadoc (April 2021): http://javadox.com/com.hermit-reasoner/org.semanticweb.hermit/1.3.8.4/overview-summary.html

[10] myCBR website (April 2021): http://mycbr-project.org/index.

[11] SPARQL syntax reference (April 2021): https://www.w3.org/TR/sparql11-query/

[12] *Javadoc* reference (April 2021): https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

[13] MVN repository (April 2021): https://mvnrepository.com/artifact/org.apache.jena/jena-arq/4.0.0

[14] JAR download (April 2021): https://jar-download.com/artifacts/org.apache.jena/jena-arq

[15] Jena Core 4.0.0 (April 2021): https://jena.apache.org/documentation/javadoc/jena/

[16] MVN repository (April 2021): https://mvnrepository.com/artifact/edu.stanford.swrl/swrlapi/2.0.9

[17] JAR download (April 2021): https://jar-download.com/artifacts/edu.stanford.swrl/swrlapi

[18] SWRL API Javadoc (April 2021): http://soft.vub.ac.be/svn-pub/PlatformKit/platformkit-kb-owlapi3-doc/doc/owlapi3/javadoc/overview-summary.html

[19] MVN repository (April 2021): https://mvnrepository.com/artifact/org.apache.jena/jena-arq/4.0.0

[20] JAR download (April 2021): https://jar-download.com/artifacts/org.apache.jena/jena-arq

[21] O'CONNOR Martin Joseph, DAS Amar. *SQWRL: A Query Language for OWL.* Stanford Center for Biomedical Informatics Research, 2009.

[22] GitHub repository (April 2021): https://github.com/CommonCoreOntology/CommonCoreOntologies