Erasmus Mundus
master course in
Data Mining and
Knowledge Management
a european master

Ontology and Semantic Web: Laboratory Work

Maíra Machado Ladeira

Lecturer: Rémi LEHN

April 25, 2015

Contents

1 Introduction

The purpose of this lab work is to develop an ontology for packages using semantic web techniques and populate this ontology with Debian's packages. After the development of the ontology, reasoning techniques are used to detect the relations between packages. The subclasses debianCommunity and windowManager are created in order to detect special debian packages, developed by the debian team or with the window manger functionality. The reasoner is used on the populated ontology, aiming to detect:

- The Debian Community packages,
- The Window Manager packages,
- Conflicts of a package,
- Requirements of a package,
- Suggestions of a package,
- Recommendations of a package,
- Packages provided by a package
- Conflict packages given a list of packages to install

In order to achieve this features, two scripts were developed using Python[1] technology:

- **ontology_generator.py**: responsible for creating and populating the ontology using RDF or Notation 3. This script receives as input a Packages' file and the type of the file to be generated (RDF or n3) and returns as output the generated ontology with the correct type.
- **reasoner.py**: responsible for performing the reasoning on the packages ontology. It receives as input the ontology file to reason and the type of reasoning to be performed and returns as output a file with the reasoning results.

For simplification of the work, in order to avoid file manipulation tasks, the ontology created on Protégé[2] is also generated by the Python script.

2 Ontology

In the first step, an ontology was created to represent the packages on Protégé. The ontology defines the classes architecture, maintainer, genericPackage, debianPackage, debianCommunity and windowManager. Architecture, maintainer and genericPackage are declared disjoint between each other. The structure of the package can be visualized in Fig. 1 In the second step, the object properties and data

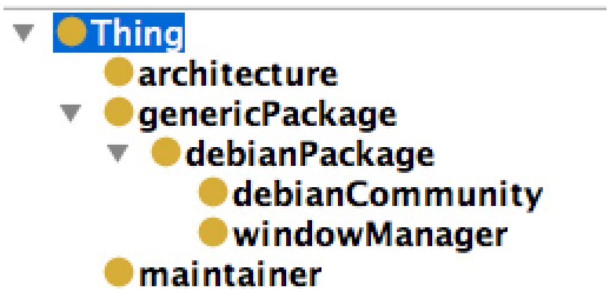


Figure 1: Classes structure for the packages ontology

type properties are set according to the required functionality where a package can have conflicts, dependencies, recommendations and suggestions and can provide another package. The Fig. 2 shows the object and data type properties structure for the ontology. The properties conflicts and depends are treated

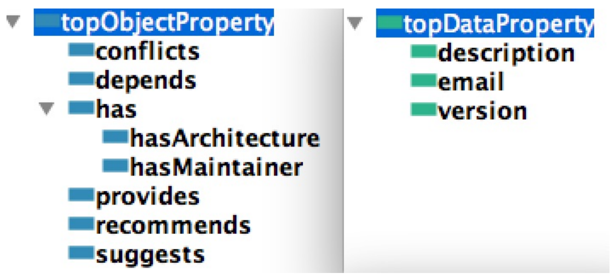


Figure 2: Object and data type properties

as transitive and the properties hasArchitecture, hasMaintainer, version and description are treated as functional.
The third and last step was to define SWRL rules to classify the debian packages into debian community or window manager packages. For this step two rules were generated:

- debianCommunity packages: are the packages with maintainer Debian_Community. The maintainer name is guaranteed to be the correct one during the parser of the ontology (described on section 2.1).

  ```
  debianPackage(?x), hasMaintainer(?x, Debian_Community) -> debianCommunity(?x)
  ```

- WindowManager packages: are the packages that have window manager as one of the functionalities. This option can be verified with the description of the package "window_manager", guaranteed during the parser of the ontology.

  ```
  debianPackage(?x), description(?x, ?y), containsIgnoreCase(?y, "window_manager")
  -> windowManager(?x)
  ```

With help of the ontology created with Protégé, the next step is to develop the Python script that generates and populates the ontology.

2.1 Ontology generator script

The script for generating and populating the ontology was developed in the programming language Python, version 3.4 and tested on the operating system OS X Yosemite[3] but should work on Windows and other unix systems as well (without guarantees). The script has two main functionalities: create the package's ontology generated on Protégé in RDF or Notation 3, according to LEHN (2011), and populate this ontology with the debian packages, described in Debian (2014), from a file provided by the user. The program was tested using the packages from the stable distribution of the main section with architecture amd64[4].

The command that automatically executes the script is described below:

```
python ontology_generator.py -p <Packages_file> -t <rdf_or_n3>
-o <Output_file> -l <limit_of_packages>
```

[1]https://www.python.org/
[2]http://protege.stanford.edu/
[3]http://en.wikipedia.org/wiki/OS_X_Yosemite
[4]Available for download on: http://ftp.fr.debian.org/debian/dists/stable/main/amd64/Packages.gz

The command parameters are explained on Table 1.

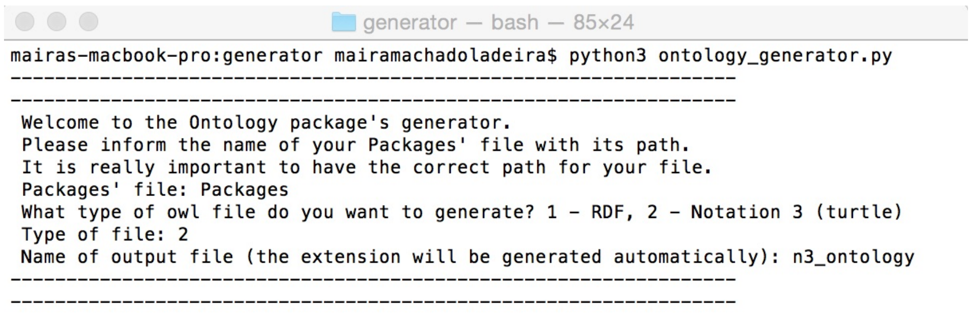
Parameter	**Explanation**
<packages_file>	file containing the packages downloaded for debian, on the URL previously described. The file should be in the same folder as the python script.
<rdf_or_n3>	type of ontology to be generated. Either rdf for RDF/XML or n3 for Notation3.
<output_file>	output file containing the generated ontology. This file should be given without the extension. The python script is responsible for setting the correct extension according to the type of the ontology.
<limit_of_packages>	Limit number of packages to be generated. This option is very useful for testing purposes but the generated packages are randomly selected. All requirement for the existence of the packages are guaranteed to be present in the generated ontology.

Table 1: Parameter options for the ontology_generator.py script.

Example of valid command to use the script:

```
python ontology_generator.py -p Packages -t rdf -o rdf_ontology
```

The program also supports the calling without parameters in this case, a command line interface guides the user to provide the necessary information during the execution time. This interface is visualized on Fig. 3.



```
mairas-macbook-pro:generator mairamachadoladeira$ python3 ontology_generator.py
-----------------------------------------------------------------
-----------------------------------------------------------------
 Welcome to the Ontology package's generator.
 Please inform the name of your Packages' file with its path.
 It is really important to have the correct path for your file.
 Packages' file: Packages
 What type of owl file do you want to generate? 1 - RDF, 2 - Notation 3 (turtle)
 Type of file: 2
 Name of output file (the extension will be generated automatically): n3_ontology
-----------------------------------------------------------------
-----------------------------------------------------------------
```

Figure 3: Ontology generator command line interface

The name of the packages, their descriptions, versions and maintainers are changed in order to follow the requisites of the ontology formats. The list of changes is presented below:

- information between () or ¡¿ are removed.
- / is replaced by _or_
- & is replaced by _and_
- The character + is replaced by _
- the characters ", ' and   are removed.
- the characters . and space are replaced by _

The result file has the extension .owl for RDF/XML ontologies and .ttf for Notation 3 (turtle) ontologies. For simplification reasons, the version of dependencies of the packages are not taken in consideration and only the important fields of the packages are translated into the ontology. Dependencies of package1 or package2 are translated to package1 and package2.

The execution time of the script, varies between 6 and 10 seconds, depending on the parallel activities performed by the computer. Feedback messages of the performed tasks are given during the whole process of execution of the script.

3 Reasoner

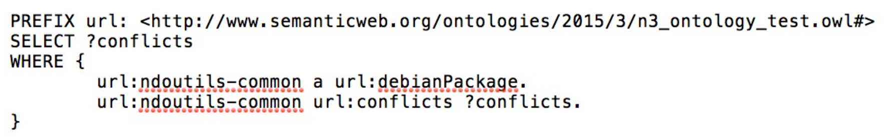
During the reasoner part of the project, the generated ontology is used to infer logical consequences on the packages. This functionality can be very useful for the analysis of the content in the packages. With reasoning it is possible to obtain, for instance, all the packages that are recommended by a package or all the packages that are required by a package. Inconsistencies and unsatisfiabilities on the ontology are also detected by the reasoner which helped on the debugging process of the ontology generator script. Query languages like sparql and N3QL are auxiliary to the reasoner process and allow more structured data manipulation during the reasoning process. For the lab work, the Pellet reasoner in version 2.3.1 and the query language: sparql, according to LEHN (2015) and Sparql (2013) were used.
A script that integrates the reasoner in order to automatically generate queries was developed in Python and will be defined on section 3.1

3.1 Reasoner script

A python script was developed in order to automatize the sparql query generation based on actions requested by the user. The script was developed in Python version 3.4 and tested on the operating system OS X Yosemite. As the system access and executes pellet on command line, this script might not work in Windows operating system (even through a fallback was implemented but not tested), but it should work on other unix systems.
The script can perform the following reasoning:

- get the packages with debian community type
- get packages with window manager type
- get the conflicts of a package
- get the packages that depends on a given package
- get the suggestions of a package
- get the recommendations of a package
- get the packages provided by a package
- get the list of packages that conflict with a given list of packages.

An example query generated for the conflicts detection can be seen on Fig. 4



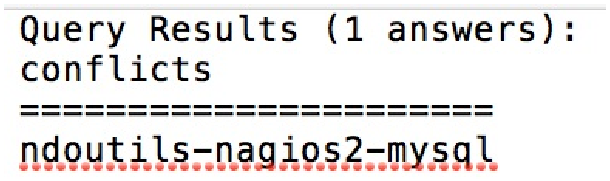
```
PREFIX url: <http://www.semanticweb.org/ontologies/2015/3/n3_ontology_test.owl#>
SELECT ?conflicts
WHERE {
        url:ndoutils-common a url:debianPackage.
        url:ndoutils-common url:conflicts ?conflicts.
}
```

Figure 4: Query generated by the reasoner script for detect the conflicts of the package: "fonts-sil-andika"

The results are saved in a file selected by the user. Fig. 5 shows an example result file for the packages that conflicts with a given package.
The script can be executed with the following code:

```
python reasoner.py -of <ontology_file> -q <query_type> -qo <query_options>
-ot <output_file>
```

The parameters options for this command are explained in Table 2.



```
Query Results (1 answers):
conflicts
=====================
ndoutils-nagios2-mysql
```

Figure 5: Results file of the reasoner script for the package "nordugrid-arc-doc"

Parameter	Explanation
<ontology_file>	The file containing the ontology to be analyzed, with extension. Use .owl for RDF/XML ontologies and .ttl for Notation3.
<query_type>	The type of the query to be executed. Should be one of the following: - window_manager, - debian_community, - conflicts_single_pack, - pack_dependencies, - pack_suggestions, - pack_recommendations, - pack_provides, - conflicts_list_of_packs
<query_options>	This option is not necessary when the query_type is window_manager or debian_community. This parameter sets the packages to be analyzed by the query. Can be a single package or a list of packages separated by comma, when query_type is conflicts_list_of_packs
<output_file>	The name of the output_file containing the answers that is generated by the reasoner.

Table 2: Parameter options for reasoner.py

A help command is also available:

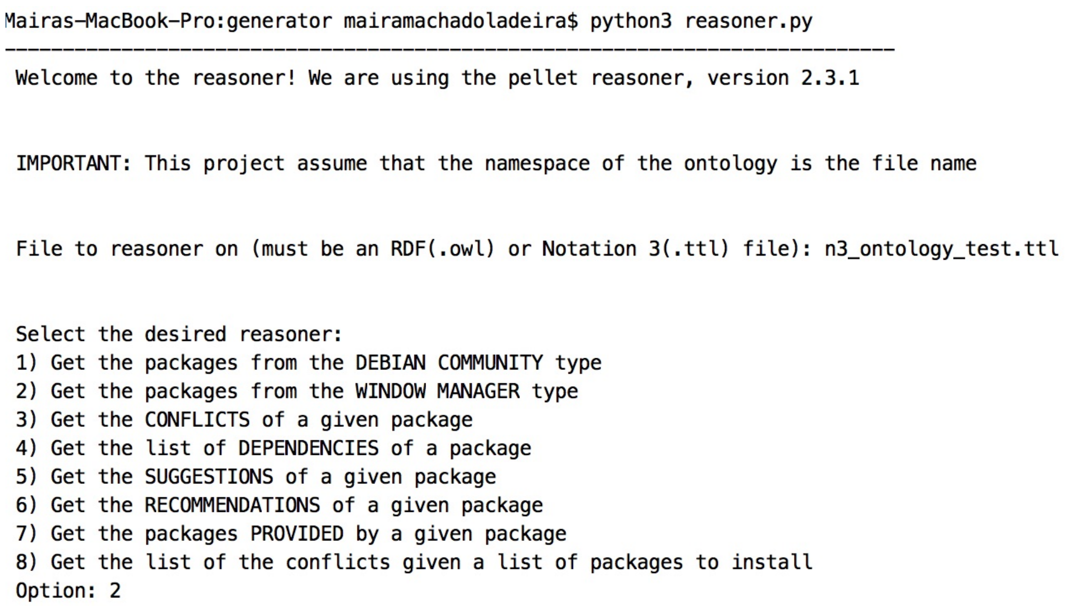
```
python reasoner.py -h
```

The program can also be executed without any parameter and in this case an interface helps the user to provide the necessary information to the reasoner. The interface of the program is visualized on Fig. 6. The script automatically parses the name of the packages in order to fit the modifications done by the ontology generator script.
For running the reasoner, a manual set up of 7G of ram for the Java virtual machine of Pellet was performed and this configuration should be changed on computers that do not support it. Due to the size of the ontology, the computer running the reasoner should have a considerable big about of RAM. Smaller ontologies (created with the -l option of the ontology generator script) can be used for testing of the program in computer with not enough RAM memory.

3.2 Sparql Queries

This section will describe the queries generated by the reasoner script. All the queries follow the basic structure defined below:

```
PREFIX url: <the_ontology_identification>
SELECT ?vars_to_be_selected
WHERE {
    #All conditions here
}
```



```
Mairas-MacBook-Pro:generator mairamachadoladeira$ python3 reasoner.py
---------------------------------------------------------------------------
 Welcome to the reasoner! We are using the pellet reasoner, version 2.3.1


 IMPORTANT: This project assume that the namespace of the ontology is the file name


 File to reasoner on (must be an RDF(.owl) or Notation 3(.ttl) file): n3_ontology_test.ttl


 Select the desired reasoner:
 1) Get the packages from the DEBIAN COMMUNITY type
 2) Get the packages from the WINDOW MANAGER type
 3) Get the CONFLICTS of a given package
 4) Get the list of DEPENDENCIES of a package
 5) Get the SUGGESTIONS of a given package
 6) Get the RECOMMENDATIONS of a given package
 7) Get the packages PROVIDED by a given package
 8) Get the list of the conflicts given a list of packages to install
 Option: 2
```

Figure 6: Interface of the reasoner script

The ontology identification, variables to be selected and conditions are defined according to the ontology and the selected type of query and are automatically built into the query by the reasoner.

- **debian community packages query:**

  ```
  PREFIX url: <http://www.semanticweb.org/ontologies/2015/3/rdf_ontology.owl#>
  SELECT ?debian_community_packages
  WHERE {
    ?debian_community_packages a  url:debianCommunity.
  }
  ```

  In this case, only the type of packages is set, once the packages are automatically inferred to be of this type with the rules.

- **window manager packages query:**

  ```
  PREFIX url: <http://www.semanticweb.org/ontologies/2015/3/n3_ontology.owl#>
  SELECT ?window_manager_packages
  WHERE {
    ?window_manager_packages a url:windowManager.
  }
  ```

  This case is similar to the debian community case but returns the packages belonging to the class windowManager.

- **Conflicts of a given package:**

```
PREFIX url: <http://www.semanticweb.org/ontologies/2015/3/n3_ontology.owl#>
SELECT ?conflicts
WHERE {
  url:libacl1 a url:debianPackage.
  url:libacl1 url:conflicts ?conflicts.
}
```

In this case, the query represents the conflicts for the package libacl1. All packages should be of type debianPackage.

- **Dependencies of a package:**

```
PREFIX url: <http://www.semanticweb.org/ontologies/2015/3/rdf_ontology.owl#>
SELECT ?dependencies
WHERE {
  ?dependencies a url:debianPackage.
  url:asterisk-mp3 url:depends ?dependencies.
}
```

Same case of conflicts option. This query returns the dependencies of the package asterisk-mp3

- **Suggestions of a package:**

```
PREFIX url: <http://www.semanticweb.org/ontologies/2015/3/n3_ontology.owl#>
SELECT ?suggestions
WHERE {
  url:acpi-support a url:debianPackage.
  url:acpi-support url:suggests ?suggestions.
}
```

Same case as conflicts and dependencies. This query returns the suggestions of the package acpi-support.

- **Conflicts between a list of packages to be installed:**

```
PREFIX url: <http://www.semanticweb.org/ontologies/2015/3/rdf_ontology.owl#>
SELECT DISTINCT ?pack1 ?pack2
WHERE {
  ?pack1 a url:debianPackage.
  ?pack2 a url:debianPackage.
  ?pack1 url:conflicts ?pack2.
  FILTER(?pack1 in (url:asterisk-mp3, url:asterisk-mysql, url:libspeex1)
  && ?pack2 in (url:asterisk-mp3, url:asterisk-mysql, url:libspeex1)).
}
```

In this query, pack1 should conflict with pack2 and they should both be present on the list of packages provided by the user. As the conflicts property is transitive, the query is able to check all the conflicts between the packages to be installed.

The queries for recommendations of a package and the packages provided by a package were omitted for being really similar to the Suggestions, Dependencies and Conflicts of a single package.

4 Conclusion

This project gave me the opportunity to learn more about semantic web and better understand the differences between the formats OWL, RDF and Notation 3. It also helped me to work with a practical use of ontology to describe real big data. The SWRL rules allowed me to understand how the classification in an ontology can be done though inferences which can be very useful in other situations like the classification of products based on their description.

This project was also important to understand and apply reasoner techniques and queries to perform logical inference in a ontology. This functionality of semantic web can be used in several situations and was well exemplified with the query to define conflicts between a list of packages to be installed.

In a further work, the tools developed can certainly be improved, incorporating for instance, the generation of ontology on the OWL format. However, the current status provides a good comparison between RDF and Notation 3 on properties like file size and time to reason on the ontology. It is also a good exemplification of the use of pellet for reasoning on big ontologies.

For technical reasons, pellet was used in an old version, since the current status generates compilation errors on Mac. The developers of pellet are aware of the problem, but it is still an open issue.[5]

All the code developed for this project is available in GitHub[6]

[5] https://github.com/Complexible/pellet/issues/16
[6] https://github.com/mairaladeira/owlpackagesgenerator

References

Debian 2014
Debian: *Debian Policy Manual Chapter 7 - Declaring relationships between packages.* `https://www.debian.org/doc/debian-policy/ch-relationships.html`. Version: 2014

LEHN 2011
LEHN, Rémi: *RDF, OWL and Semantic Web.* 2011. – DMKM Slides

LEHN 2015
LEHN, Rémi: *SPARQL.* 2015. – DMKM Slides

Sirin et al. 2007
Sirin, Evren; Parsia, Bijan; Grau, Bernardo C.; Kalyanpur, Aditya; Katz, Yarden: Pellet: A practical owl-dl reasoner. In: *Web Semantics: science, services and agents on the World Wide Web* 5 (2007), Nr. 2, pages 51–53

Sparql 2013
Sparql: *SPARQL 1.1 Overview.* `http://www.w3.org/TR/sparql11-overview/`. Version: 2013