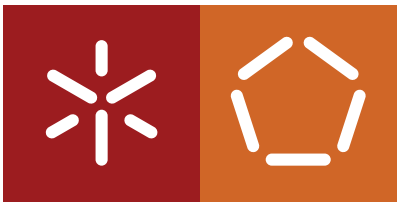

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Master Course in Computing Engineering

João Manuel Sousa Fonseca

Converting ontologies into DSLs

Master dissertation

Supervised by: Pedro Rangel Henriques
Maria João Varanda

Braga, September 22, 2014

PARECER

Serve o presente parecer para declarar que o aluno *João Manuel Sousa Fonseca* concluiu, conforme esperado, a escrita do seu relatório de pré-dissertação. O documento foi revisto pelos orientadores, os quais atestam a sua validade científica, assim como o cumprimento dos objetivos propostos para esta etapa. Mais se informa que as atividades de mestrado do aluno *João Manuel Sousa Fonseca* decorrem dentro dos planos e prazos inicialmente previstos.

Pedro Rangel Henriques	*Maria João Varanda*
(Orientador)	(Co-orientador)

ACKNOWLEDGEMENTS

First of all I will like to express my gratitude to PhD. Pedro Rangel Henriques and PhD. Maria João Varanda for the magnificent support and dedication to this project.

In addiction, I big thanks to my mother, father and little brother for being always be my side and supporting me in every decision that I make.

A very special thank you to all my grandparents; João Sousa and Helena Vilas Boas, João Fonseca and Angelina Sousa; for all the affection and dedication because even when I try to explain my work and they don't understand, they support me.

Finally but not last, I like thankfully for my few but good closest friends; Sara Viera, Sofia Oliveira, Rita Faria, Cátia Félix, Mário Esteves, Jorge Soares and Diogo Vieira; that are always ready to help and give me motivation to move on.

ABSTRACT

This paper presents a project whose main objective is to explore the Ontological-based development of Domain Specific Languages (DSL), more precisely, of their underlying Grammar.

After reviewing the basic concepts characterizing Ontologies and Domain-Specific Languages, we introduce a tool, *OWL2Gra*, that takes profit of the knowledge described by the ontology and automatically generates a grammar for a DSL that allows to discourse about the domain described by that ontology.

This approach represents a rigorous method to create, in a secure and effective way, a grammar for a new specialized language restricted to a concrete domain. The usual process of creating a grammar from the scratch is, as every creative action, difficult, slow and error prone; so this proposal is, from a Grammar Engineering point of view, of uttermost importance.

After the grammar generation phase, the Grammar Engineer can manipulate it to add syntactic sugar to improve the final language quality or even to add semantic actions.

The *OWL2Gra* project is composed of three engines. The main one is *OWL2DSL*, the component that converts an OWL ontology into an attribute grammar. The two additional modules are *Onto2OWL* and *Ddesc2OWL*. The former, *Onto2OWL*, converts ontologies written in OntoDL (a light-weight DSL to describe ontologies) into standard OWL XML that can be loaded into the well known Protégé system to future editing; the later, *Ddesc2OWL*, converts domain instances written in the DSL generated by OWL2DSL into the initial OWL ontology.

Ddesc2OWL plays an important role because it allows for the population of the original ontology with concept and relation instances extracted from the new language concrete sentences this allow a faster ontology population.

CONTENTS

LIST OF FIGURES

LIST OF LISTINGS

1

INTRODUCTION

The use of domain-specific languages (DSL) has increased in the recent years. This technology enables a quick interaction with different domains, thereby taking a greater impact on productivity because there is no need for special or deep programming skills to use that language. However, to create a domain-specific languages is a thankless task, which requires the participation of language engineers, which are (usually) not experts in the domain for which the language is targeted Robert Tairas (2009). Therefore, the participation of domain experts in this process is also commonly required.

The experts' task is to organize the domain knowledge in such a way that the language engineer is able to incorporate the domain concepts in the concepts of the language. Although the latter is known as a complex and time consuming task, it brings together the program and the problem domains of the DSL, which is one of its most important characteristics Oliveira (2010).

The domain knowledge is usually organized in ontologies. Informally, an ontology is an artifact that defines a set of concepts, relations and axioms for a specific knowledge domain. It represents and organizes the implicit knowledge in such a way that a set of cooperative systems agree on it and share D.Jin (2004) Grimm (2010). In practice, ontologies are usually represented in OWL files (a particular XML dialect) of very easy comprehension, but not so easy creation. Fortunately, there are tools such as *Protégé*[1], that help on this process.

Also the creation of population for these domains is a thankless work. This was one of the last priorities of this project. The idea behind has to gather the domain specification of the DSL and using that DSL to generating inputs for the ontology this process would allow a faster ontology generated population, once the creation of individuals one by one on the ontology is a hard work.

1.1 OBJECTIVES

The work hereby proposed aims at taking advantage of the processable nature of OWL ontologies to generate DSLs from the enclosed domain knowledge. This is expected to automatize, at a certain extent, the language engineer task of bringing program and problem domains together.

1 http://protege.stanford.edu/

Ontologies are usually created as only a scheme of the domain knowledge. But this is far from being a complete ontology. Ontologies also support instances of the concepts and their relations, but populating such *database* is a tremendous manual and time consuming routine.

A foreseen byproduct of the proposed work is the possibility of populating ontologies from text files written in the new and automatically generated DSL. This would combine the best of both worlds. In practice, it is desired to take advantage of modeling the domain as an ontology from where a DSL (and its processor) can be extracted. The DSL processor can be specialized to convert its input (the programs) into new OWL files containing the instances capable of being extracted from key parts of the program.

1.2 RESEARCH HYPOTHESIS

Given an abstract ontology, describing a knowledge domain in terms of its concepts and the relations among them, it is possible to derive automatically a grammar to define a DSL for that same domain.

1.3 DOCUMENT STRUCTURE

This document describes the work that has been done for the last several months and will be divide in seven important chapters.

In the second chapter, it will be presented a brief description about the state of the art and it will be characterized in more detail the main technologies presented in this project: ontologies and domain specific languages.Other existing tools that were developed with one or both technologies, will be referred.

In third chapter, it will be presented the architecture of the system proposed, OWL2GRA, to solve the problem. A general overview of OWL2GRA will be provided. Each one of the three system components will be described.

Chapter four describes Onto2OWL. This module accepts the description of and ontology in OntoDL, a DSL we have defined to allow an easy and light description, and convert it into OWL standard. The generated OWL file can be loaded into a tool like Protégé to allow further processing.

The fifth Chapter describes the heart of the project, the module OWL2DSL.This module converts ontologies into Domain Specific Languages(DSL's).The result is a grammar with syntactic sugar and semantic axioms. Several other files that help and reduce work for the users are also generated. This Module can be used separately from the previous one because it accepts all the ontologies that are specify on all standard formats.

In Chapter six it will be described Module Ddesc2OWL that was not on the initial list of objectives but soon became one priority of this project, because of the possibilities that can bring to the generation of the individuals of an ontology. It was noticed that the generation of the grammar that will describe the domain will create individuals and actions for that same domain. The idea was to gather that

individuals and actions and populate the ontology this allow the faster creation of individuals from any domain that was specified from an ontology. The ontology populated also allows an important functionality that is the ontology querying, this feature can be used on multiples situations, such as a small database using SPARQL or even using it on web-semantic, that helps with page indexation and better querying results from the search engines on the web.

Chapter seven is the Conclusion, where the results of this master thesis will be explained as well as directions for future work. Also and not less important, the problems that appeared and the way they were solved will be discussed.

This document also includes three Appendixes where some more realistic and different examples will be presented in order to explain with more detail the results of OWL2GRA.

2

STATE OF THE ART

In this master project, it will be used technologies like Domain Specific Languages and Ontologies and in the sections bellow, it will be described some characteristics of this technologies and some projects where they were used.

2.1 ONTOLOGIES

Ontologies can be used in many differently situations, even they can be used for different purposes. They also can be split according its degree of complexity, expressivity of the domain and formality. This technology is normally associated with Web-Semantic with two main objectives, describe objects or categorize them. In nowadays the ontologies, are used to catalog the websites that appear on the results from search engines like Google, Bing or Yahoo, also can be used to make direct queries to the network of knowledge granted by ontologies, for example, DBpedia.[1].

2.1.1 *The Hermes Project*

This project has the purpose to process data systems into knowledge systems. This is for a better understanding of things like properties of autonomy, sociability and learning abilities of software. The knowledge systems provide better usability and effectiveness than tradicional systems. The ontologies were used to representing the knowledge bases, because the reusable and easy extension. This project also infers the taxonomic relationships between concepts is very useful to define hierarchies and to give meaning to relations between concepts. Girardi (2010)

2.1.2 *Lightweight Ontologies*

The idea behind this project is to categorize objects and trying to resolve the problem of classifying, for example, photos, webpages or books. Lightweight uses ontologies with tree structure where each node is associated with a natural language label. Also, sometimes Formal Lightweight Ontologies are obtain from Lightweight Ontologies by translating the meaning of the labels with Description Logics

1 http://dbpedia.org/About/

formulas witch captures the label meaning and provides an example of how such translation can be done.In Formal Lightweight Ontologies, the node formulas are describe in a subsumption relation, so the formula is always more general in the father node then the formula on the child node. Another point in favor for the Lightweight Ontologies is the automated documentation, classification or query answering. Even with advantages, this technology does not have the support from the users because among other problems is the lack of interest of the user to build the Ontologies. Giunchiglia et al. (2009)

2.2 DOMAIN SPECIFIC LANGUAGES

A Domain Specific Language is a tailor made notation toward a specific domain, this means that it is much easier to describe and generate information relative to that domain. (Kosar et al., 2008) Also with DSL, the productivity can be increased because the knowledge of the domain is less require. The downfall of this technology is that is very expensive to generate, because to generate a DSL the programmer must have some expertise on both domain and coding language.

2.2.1 *The IDEA project-Implementation of DSL: Evaluation of Approaches*

This project was born from the ideia of putting all the approaches to DSLs implementations together. The result was a detailed description of each approach and also how hard is to implement the approach. With this study it was able to understand that the approach with more efficient result was the *Embedded Implementation*.

The *Embedded* approach, consist in taking, the existing mechanism in the native language that is used to create the DSL. Then the semantic tools are used to specify the domain then the embedded DSL receive the language constructs and add the domain specification from primitives that are closer to the user DSL. The main limitations notice in this approach is the syntactic mechanism in the language. Also, it was notice that the error reporting is problematic because the messages are in terms of the language concepts.

In this paper was also tested the *Preprocessing* approach, this technique aggregates the DSL constructs and translate them to the native language, but with this, the static analysis is limited.

Another method is the *Compiler Generator*. This approach is similar to the *Embedded*, excepts that some of the work of compiling is done using the native language with tools. (Kosar et al., 2008)

2.2.2 *Feature Description Language*

This paper address the feature diagrams in more detail, thus as relationship with a DSL. For that, it was developed the Feature Description Language (FDL), a textual language to describe features. This technology, explores automated manipulations of features descriptions like normalization, expansion

to normal form, variability computation and constraint satisfaction. This project was divided in four main phases.

First of all , the formalization of the notion of feature diagrams, by using DSL for features definitions called FDL. This also allow the manipulation of FDL expressions.

Secondly, to support feature was developed a prototype tool and get new ideas for further development.

Another objective, has to resolve the problem that was difficult to find actual examples of features diagrams used in concrete projects.

Finally, was unclear how to proceed once a diagram exist. To address this problem was created a tool that does the mapping of an FDL description to a UML diagram that allow a first version of the configuration interface. van Deursen and Klint (2001)

2.3 CONVERTING ONTOLOGY TO DSL

A Domain Specific Language has a detailed knowledge about a certain domain. However, to generate this DSL is required paradigms such as *Generative Programming* or *Domain Engineering*. The Ontology has the ability to describe any domain with precision. So, why not merge these two technologies? The works below were conceived on that same basic idea, that extracts from the Ontology the knowledge of that specific domain to generate a DSL.

2.3.1 *Using Ontologies in the Domain Analysis of Domain Specific Languages*

The meaning of this work is to achieve the domain analysis to induce what elements can be reused. This is very helpful, when it's used paradigms such as the Generative Programming or Domain Engineering. Like it was referred before, to build a DSL, it's required overall knowledge from the domain, this domain model is given by the ontology, revealing important information that influence the language shape. The ontology contributes for the early stages of the domain analysis. So, this work uses ontologies to validate the domain analysis and to get the domain terminology for the DSL creation.

In this paper, the author uses ontology to help with the domain terminology, but it can also be use to describe the concepts domain and its relationships. That reveals interesting connection between DSL and ontologies. A domain model is defined by defining the scope of the domain, the terminology, concepts description and features model describing commonalities and variabilities.

The case study on this paper show improvement of the DSL development using ontologies on the early stages. Because, an ontology can provide a defined and structured process domain. Also with the domain specification on the ontology there is no need to start from scratch the DSL development Tairas et al. (2009).

2.3.2 *Ontology Driven Development of Domain-Specific Languages*

This paper propose an ontology-based domain analysis and its incorporation on the early design phase of the DSL.

In this article there are covered two types of languages. The type of language to which this article is directed are the DSL, Domain Specific Language, these language normally are tailored made to a specific domain. In this case,the intentions is study what is the best approach to DSL development.

In this paper, the authors identified several phases that a DSL development need. The most important are the decision, analysis, design, implementation and deployment.

The decision phase is important because defines all the answers to the questions for development of a DSL.

In the other phases, the process is very similar to the GPL development, with the difference in the activities , approaches and techniques used in each phase.

In the domain analysis, it's a process that uses several methodologies that differ from the level of formality, information extraction or their products. The objective is to select and define the domain focus and collect the important information and integrate with a domain model.

In the design phase, it has the definition of the constructs and semantics values. The semantics has the objective of formalize the meaning of the constructs and also the detail and behavior unspecified in the program. The DSL can be classified on two dimensions, the relations between the DSL and the computer language and the formality of the DSL description.

The implementation phase, as the name indicates, is the phase where was decided what approach to implement and also what approach required less effort and has more efficacy to the end user.

In the paper was created a framework to enable the automated generation of a grammar construction from a target ontology. This framework is the Ontology2OWL. This framework accepts OWL files as input and parses it and generate and fill internal data structures. Then a processor apply a set of transformation patterns over the data structures. The result is a grammar, generated automatically, that can be inspected by a DSL engineer in order to verify if there is something wrong Ceh et al. (2011).

3

OWL2GRA: ARCHITECTURE AND GENERAL OVERVIEW

The proposal for the *OWL2Gra* project here reported is to create automatically an attribute grammar for a concrete Domain Specific Language, based on an ontological description of that domain.

Figure 1 — the block diagram that depicts the architecture of *OWL2Gra* system — represents all the processes and modules that are presented in this project showing how an abstract ontology(described in a simple ontology specification language) is transformed into an OWL XML ontology that is then transformed into an AntLR attribute grammar.

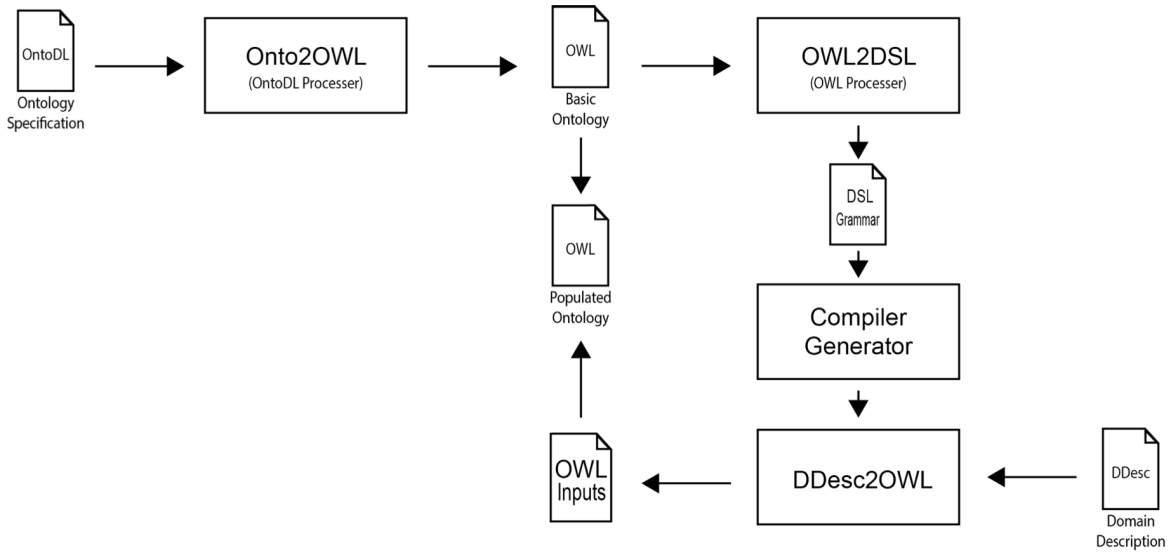

Figure 1.: Onto2Gra

The first module of this project is called Onto2OWL. This phase was designed to improve the knowledge on the problem. The idea behind this layer is to offer the possibility of conversion of a simple text and description into a OWL standard file. The original ontology shall be described in a DSL, a kind of natural language specifically tailored for that purpose, called OntoDL – Ontology light-weight Description Language. This tool, *Onto2OWL*, takes an OntoDL file, that is an ontology specification file, and converts it into OWL XML, that is the standard format for ontology descriptions. The aim of this tool is to offer an easy way to build a knowledge base to support the next phase.

However, it is important to notice that this phase is not mandatory – this step can be skipped if the source ontology is already available in OWL format (or even in RDF XML format). In that case, the user of *OWL2Gra* system can go directly to the second stage.

The second block is the most important on this proposal, and also the core of *OWL2Gra*. It is composed of a tool, *OWL2DSL* that makes the conversion of an OWL XML or a RDF XML file into an attribute grammar. This grammar is created systematically from a set of rules that will be explained in Chapter 5. From the OWL ontology description, *OWL2DSL* is able to infer: the non-terminal and terminal symbols; the grammar production rules; the symbol attributes and their evaluation rules[1]. Besides that, *OWL2DSL* generates a set of Java classes that are necessary to create an Internal Representation of the concrete ontology[2] in order to store all the information that will be processed by the generated grammar.

The Grammar generated by *OWL2DSL* is written in such a format that can be compiled by a Compiler Generator (specifically in our case we are using the AnTLR compiler generator) in order to immediately create a processor for the sentences of the new Domain Specific Language. AnTLR is very helpful because it builds a Java program to process the target language; we call that processor *DDesc2OWL* and it is precisely the engine in the center of the third block.

In that third block of the *OWL2Gra* architecture, the tool *DDesc2OWL*, will read a Ddesc input file, with a concrete description of the Domain specified by the initial ontology, and will generate an OWL or a RDF XML file that, when merged into the original OWL or RDF XML file, will originate a specification that populates the original ontology creating a network of knowledge.

In the next chapters it will be described in great detail the objectives of each phase, the problems found and how they were solved.

3.1 APPLICATION USAGE MODES

OWL2Gra system has three modules that can be used separately. The first objective of this project was to create a Java graphical interface to provide access to this three modules.

Figure 2 shows the Java Application graphical interface that offers to the end-user a simple usage mode.

1 In the future we will also be able to derive the contextual conditions.
2 An ontology with instances.

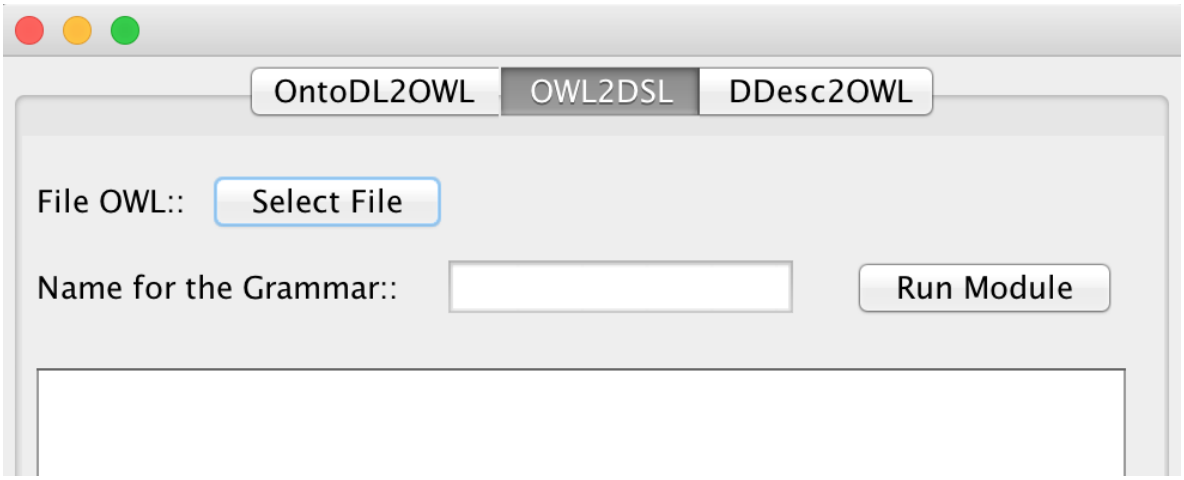

Figure 2.: Java Application OWL2Gra

This platform was created using the Java Swing libraries that help to build a simple Frame that present all the possibilities that are on the OWL2Gra program.

This program can also be used by third part using the command line interface(instead of the graphical one shown and described above.

Figure 3 shows the help menu available in the command line interface.

```
OWL2GRA_jar|master⚡ ⇒  java -jar OWL2GRA.jar help
************************* OWL2Gra ****************************
*  ->Module Onto2OWL                                         *
*     argument[0] => onto2owl                                *
*     argument[1] => OntoDL file path                        *
*     argument[2] => Name for outcome                        *
************************************************************
*  ->Module OWL2DSL                                          *
*     argument[0] => owl2dsl                                 *
*     argument[1] => OWL file path                           *
*     argument[2] => Name for outcome                        *
************************************************************
*  ->Module DDesc2OWL                                        *
*     argument[0] => ddesc2owl                               *
*     argument[1] => OWL file path                           *
*     argument[2] => Grammar file path                       *
*     argument[3] => DDesc file path                         *
************************************************************
```

Figure 3.: Help Menu from Command Line Support

4

ONTO2OWL MODULE

Module Onto2OWL is the first layer of this project. Onto2OWL creates a specification file in the standard notation for ontologies specification, OWL-XML, from an ontology specification written in OntoDL, a Domain Specific Language to describe in a simple, easy to use, way the ontologies.

This module is composed by two important parts. The first is a parser for the input file written in a DSL that we have specially designed, called *OntoDL*. The second part is a java class that manipulates the information gathered by the parser and generates the OWL-XML standard file.

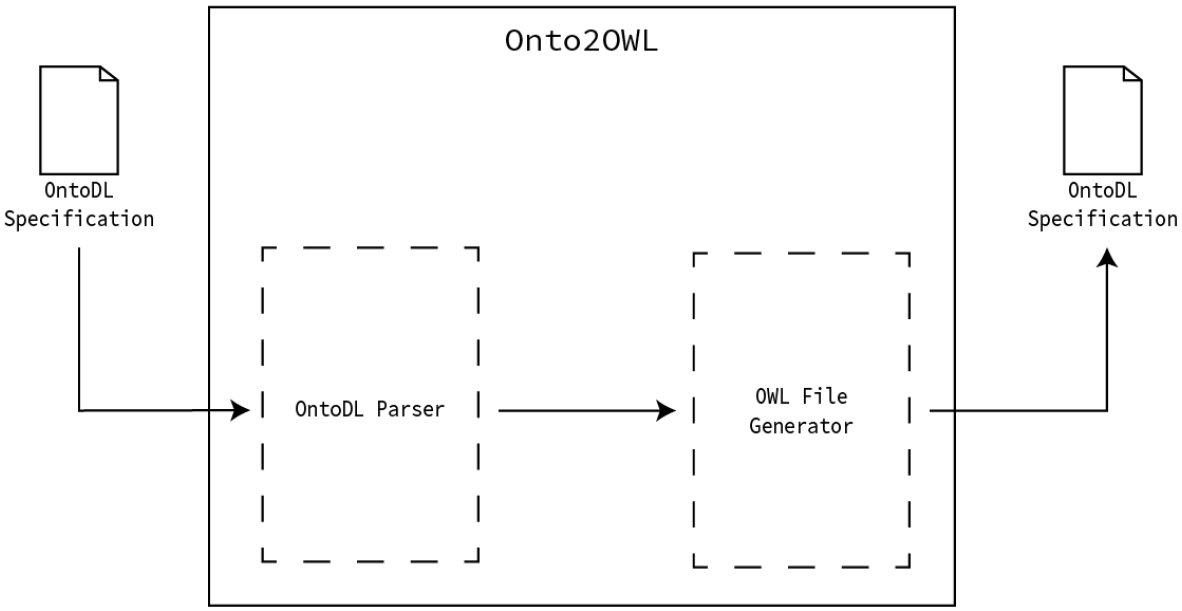

Figure 4.: Onto2OWL Architecture

In the next subsections it will be explained how these two parts work together.

4.1 THE PARSER FOR ONTODL FILES

This parser is generated from a grammar that was created to specify OntoDL language.This parses is generated based on an attribute grammar that collects all the informations about the domain and returns it to the main program that invoques the Parser.

This attribute grammar specifies the syntax and the semantic of the language OntoDL that was define to specify ontologies; it is a quite simple grammar.

This parser will recognize all the basic components of an ontology described in OntoDL language. After analyzing the problem, it was decided that only four components are necessary to describe the basic information about an ontology: *Concepts*, *Hierarchies*, *Relations* and *Links*. So, the first grammar rule is:

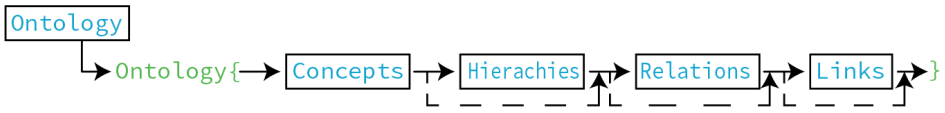

Figure 5.: "Ontology" production

This explain how simple is to describe a domain though an ontology. In listing 4.1 bellow is an example how this first part is written in OntoDL.

```
Ontology{
    Concepts[...]
    Hierarchies[...]
    Relations[...]
    Links[...]
}
```

Listing 4.1: OntoDL Example: Ontology specification

An ontology is a description of a certain domain. In order to describe the domain objects the ontology uses *Concepts*, or Classes. A *Concept* has a name, a description and a list of attributes.

An *Attribute* is a field that defines a characteristic of a *Concept*. It is composed of a name and a type that can be a 'string', 'int', 'boolean' or 'float'. Figure 6 represents all these productions.

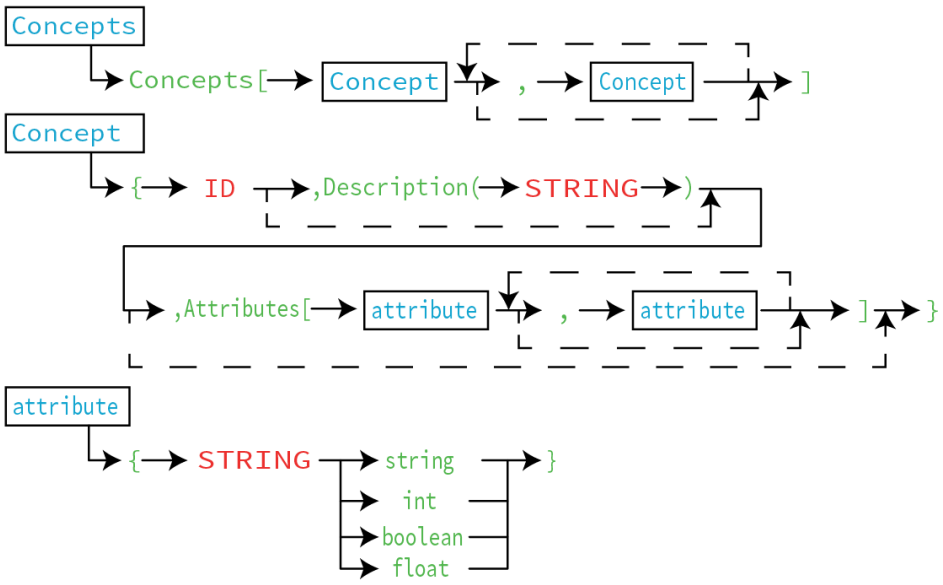

Figure 6.: 'Concepts' productions

This part is important because it declares all the Classes that will be present on the ontology and that will be used on the following specifications like *Hierarchies* and *Links*. The listing bellow represent an example of the two *Concept* specifications.

```
Concepts[
    {
        DSL,
        Description("tailor made notation toward a specific domain")
        Attributes[
            {"attribute_1" string}
            {"attribute_2" int}
        ]
    },
    {
        ProgramLanguage
    }
]
```

Listing 4.2: OntoDL Example: Concept specification

After the *Concept* specification, it is possible to define the hierarchy between two *Concepts*, the first is the super-class *Concept* and second is the sub-class *Concept*. If one of the *Concepts* is not previously specified, the program ignores the *Hierarchy* that is being specified and issues a warning message. The production for the *Hierarchy* specification is represented in Figure 7.

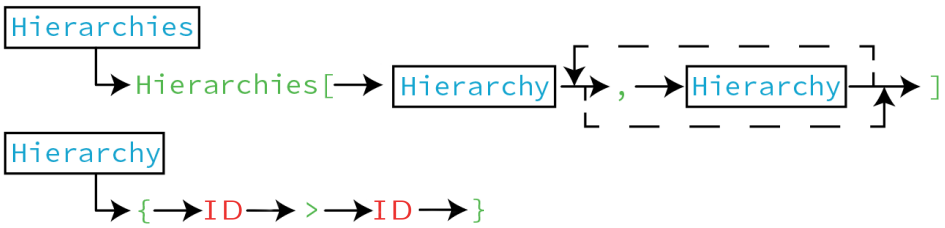

Figure 7.: “Hierarchies” production

The *Hierarchies* production specifies the relations there are normally represented by the link “is-a”. Listing 4.3 is a *Hierarchy* example extracted from a OntoDL file that says that Concept “DSL”, the subclass, is-a “ProgramLanguage”,the super class.

```
Hierarchies[
    {
        ProgramLanguage > DSL
    }
]
```

Listing 4.3: OntoDL Example: Hierarchy specification

After defining the hierarchical relations holding among *Concepts*, it is necessary to define the non-hierarchical *Relations* that will be used to connect *Concepts*. A *Relation* is a bridge between *Concepts* and it brings semantic value to the domain graph. With that in mind it was added a production rule to describe a *Relation*, as shown in Figure 8.

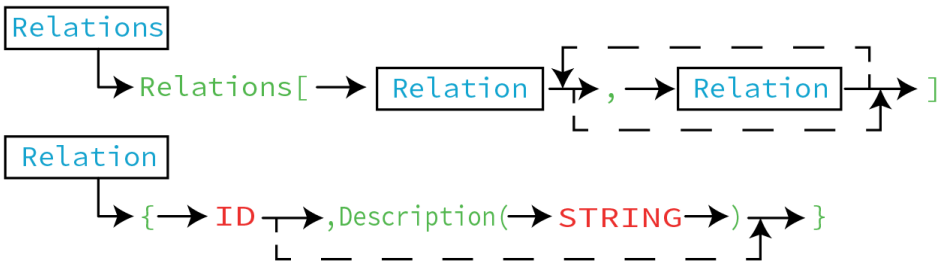

Figure 8.: “Relations” production

This group of *Relations* include all the other relations that link Concepts and are not the hierarchical “is-a” connection. Listing 4.4 bellow shows how to specify this part.

```
Relations[
    {
        bases
    },
    {
        produces
    }
]
```

Listing 4.4: OntoDL Example: Relation specification

At last, we need to define the *Links* that are used to identify the *Concepts* and the *Relation* that connect them.

A *Link* is composed of two *Concepts* and one *Relation*. If one of the three items is not previously specified, the *Link* is ignored and a warning is displayed on the console.The first *ID* that appears is the *ID* for the first *Concept*, the next is for the relation and finally the last is for the second *Concept*.

Another interesting part about the Links is the possibility to specify cardinality constraints between the Concepts. This allows a more precise and correct grammar generation as well as better domain specification. There are of course, two cardinality options to specify.

The first is the "min" constraint, this allows the user to specify the minimum number of Links that will be needed to create.

The second is clearly the "max" constraint, this allows the user to define the maximum number of Link that can be created.

The important thing about this cardinality properties is that both are not required and the specification of one of them does not required the specification of the other. However, if none of this cardinalities are specified the generation will assume that the Link is an *N-M* relation.

The *Links* have the specification that is shown in Figure 9.

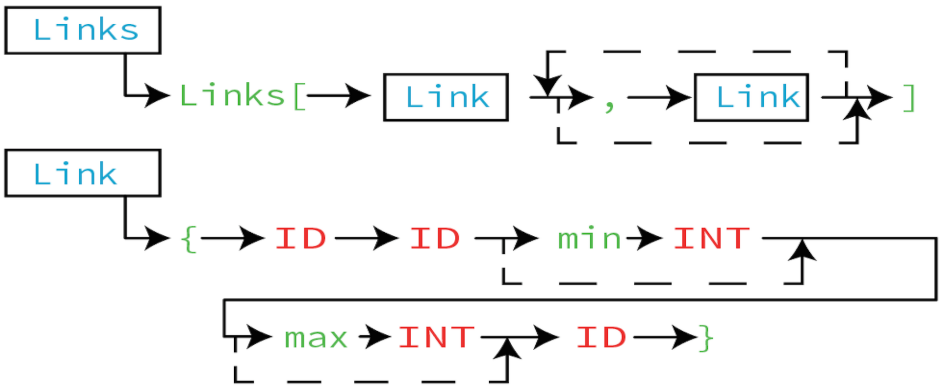

Figure 9.: "Links" production

This language construction specifies the *Relation* that connects the *Concepts*. Listing 4.5 bellow illustrates how to specify a link.

```
Links[
    {
        KnowledgeDomain bases Languagedesign
    },
    {
        Languagedesign is_expressed Grammar
    },
    {
        Grammar produces min 1 max 1 ProgramLanguage
    }
]
```

Listing 4.5: OntoDL Example: Link specification

In conclusion, with this grammar specification it is possible to generate a parser to process an OntoDL specification. This parser will store the information extracted from the input file into a set of Java classes that were designed to accommodate the needs of the translator, as explained in the next subsection.

4.2 THE OWL FILE GENERATOR

The OWL Generation uses the information that were retrieved by the parser and stored in the Java classes in order to generate the final product that is an OWL XML file.

The parser returns an object of the class *Ontologies*. Listing 4.6 shows how this class is organized.

```
public class Ontologies{
    public ArrayList<Concepts> concept;
    public ArrayList<Hierarchies> hierarchy;
    public ArrayList<Relations> relation;
    public ArrayList<Triples> triple;
    public void gerarowl(String input) throws FileNotFoundException {
      ...
    }
}
```

Listing 4.6: Ontologies Class

As the listing above shows, the object that is returned by the parser already has all the information required to generate the OWL file. This information is stored in four important Array List.

However before going into details about these four classes and explaining the OWL generation derived for each one, it is necessary to describe the beginning of the file that must be generated. Any OWL specification starts with the references to the standard notations used. The listing bellow shows the start specification for the OWL files.

```
<?xml version="1.0"?>
<!DOCTYPE Ontology [
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY xml "http://www.w3.org/XML/1998/namespace" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>
<Ontology xmlns="http://www.w3.org/2002/07/owl#"
     xml:base="http://example.com/onto2owl/langprocessor#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:xml="http://www.w3.org/XML/1998/namespace"
     ontologyIRI="IRI for the langprocessor">
    <Prefix name="rdf" IRI="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
    <Prefix name="rdfs" IRI="http://www.w3.org/2000/01/rdf-schema#"/>
    <Prefix name="xsd" IRI="http://www.w3.org/2001/XMLSchema#"/>
    <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
```

Listing 4.7: OWL Specification header

The first array stores all the *Concepts* that are identified by the parser. The *Concepts* object class is specified in Listing 4.8.

```
public class Concepts{
    public String name;
    public String description;
    public ArrayList<Atributes> atribute;
}
```

Listing 4.8: “Concepts” Class

The OWL specification for the Concepts is very simple, just a simple declaration. Listing 4.9 illustrate such a declaration for the specification exemplified in Listing 4.8.

```
<Declaration>
    <Class IRI="#KnowledgeDomain"/>
</Declaration>
<Declaration>
```

```
    <Class IRI="#Languagedesign"/>
</Declaration>
```

Listing 4.9: “Concept” Specification in OWL

The second array stores the *Hierarchies* between concepts. This object is very simple, it only saves the name of the super-class and the sub-class name, as shown in Listing 4.10

```
public class Hierarchies{
    public String class_name;
    public String subclass_name;
}
```

Listing 4.10: “Hierarchies” Class

The hierarchical connections are an important part of the ontology. The OWL that must be generated for a hierarchy is exemplified in Listing 4.11.

```
<SubClassOf>
    <Class IRI="#DSL"/>
    <Class IRI="#ProgramLanguage"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Interpreter"/>
    <Class IRI="#LanguageProc"/>
</SubClassOf>
```

Listing 4.11: Hierarchical Specification in OWL

The third array is also important because it stores all the non-hierarchical relations between *Concepts*. A non-hierarchical relation must be defined previously so that its name can be used to create *Links*. Listing 4.12 shows how the information about *Relations* is saved in a Java class.

```
public class Relations{
    public String name;
    public String description;
}
```

Listing 4.12: “Relations” Class

In OWL the relations are not specified directly. Instead of that, they are defined as object properties and so they are specified by the domain (the object of the relation) and the range (the object that is associated). So to generate the respective OWL we need the Links. As was referred above, the fourth

array is the one that stores the *Links* or *Triples*. The *Links* represent what *Concepts* are connected and with what *Relation*. The respective Java class is defined in Listing 4.13.

```
public class Triples {
    public String class1; // Super-class
    public String relation;
    public int min;
    public int max;
    public String class2; // SubClass
}
```

Listing 4.13: “Triples” Class

Another important point concerning Object Properties is that in case of multiple domains for the same property, it is important in the ranges to make a simple annotation to specify the precise domain for each range.

```
<ObjectPropertyDomain>
    <ObjectProperty IRI="#bases"/>
    <Class IRI="#KnowledgeDomain"/>
</ObjectPropertyDomain>

<ObjectPropertyRange>
    <Annotation>
        <AnnotationProperty abbreviatedIRI="owl:backwardCompatibleWith"/>
        <IRI>#KnowledgeDomain</IRI>
    </Annotation>
    <ObjectProperty IRI="#bases"/>
    <Class IRI="#Languagedesign"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
        <Annotation>
            <AnnotationProperty abbreviatedIRI="owl:backwardCompatibleWith"/>
            <IRI>#Grammars</IRI>
        </Annotation>
        <ObjectProperty IRI="#produces"/>
        <ObjectMinCardinality cardinality="1">
            <ObjectProperty IRI="#produces"/>
            <Class IRI="#ProgramLanguage"/>
        </ObjectMinCardinality>
    </ObjectPropertyRange>
```

Listing 4.14: Links Specification in OWL

4.2. The OWL file generator

To generate the final OWL file, the system just has to execute the method "RunModule(String ontofilepath, String name_for_ontology)" that belong to the class "Onto2OWL". This function executes the OWL generation using information collected and stored in the object "ontology". That is returned by the OntoDL parser.

This method receives a string parameter. This parameter specifies the name of the OWL file; normally this name is the same as the OntoDL file, sent as input to the parser.

This function computes some statistics and also gathers information about the generation of the final file. As can be seen in Figure 10, this information is displayed in the Text Area of the *OWL2Gra* interface.

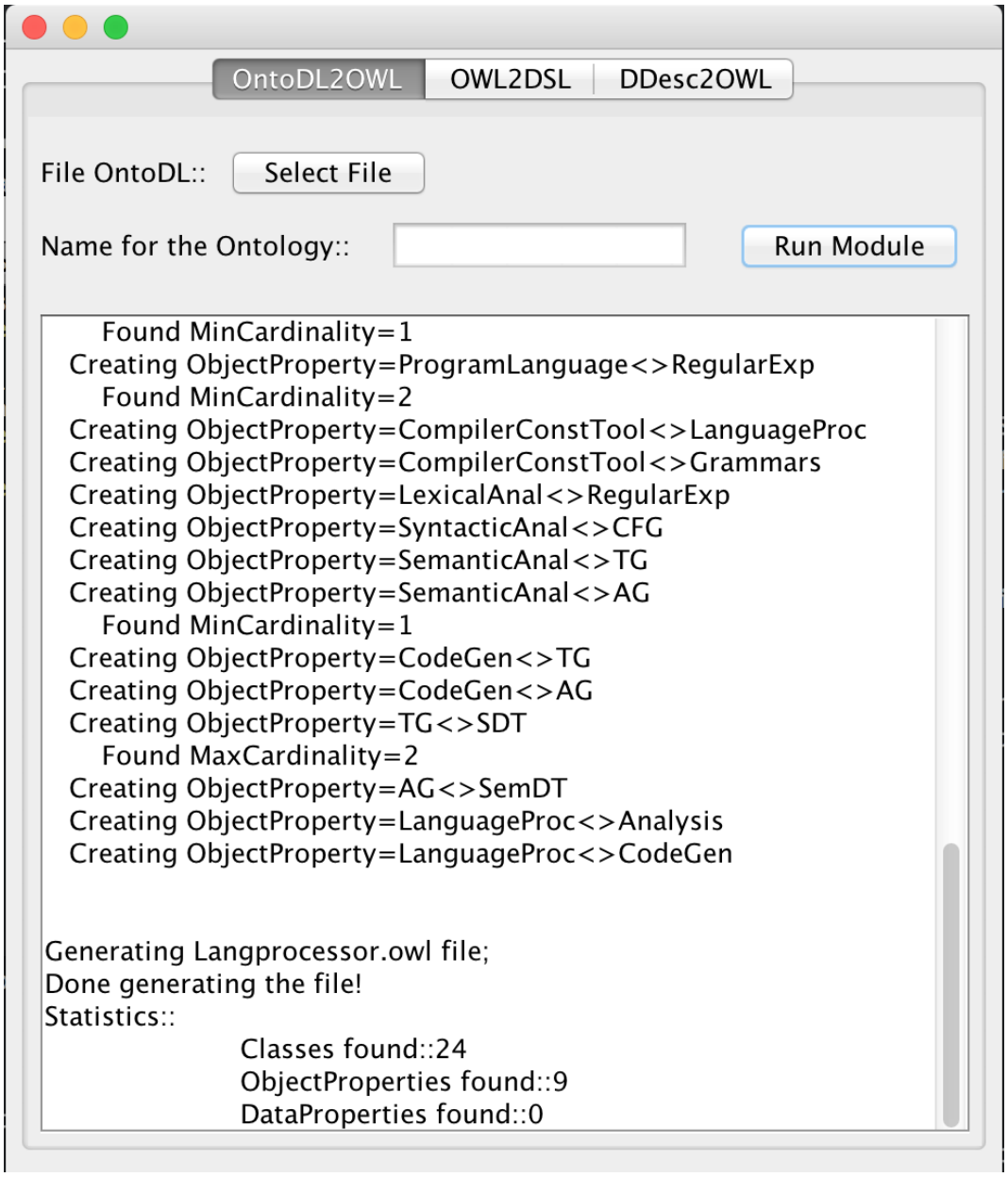

Figure 10.: Statistics processed by Onto2OWL Module

After the generation of the OWL file we can work more on the ontology, to explore or to edit it.For that purpose we can load it into Protégé[1], as displayed in Figure 11. This allows us to add information to the domain without creating specifications from the scratch.

1 http://protege.stanford.edu/

4.2. The OWL file generator

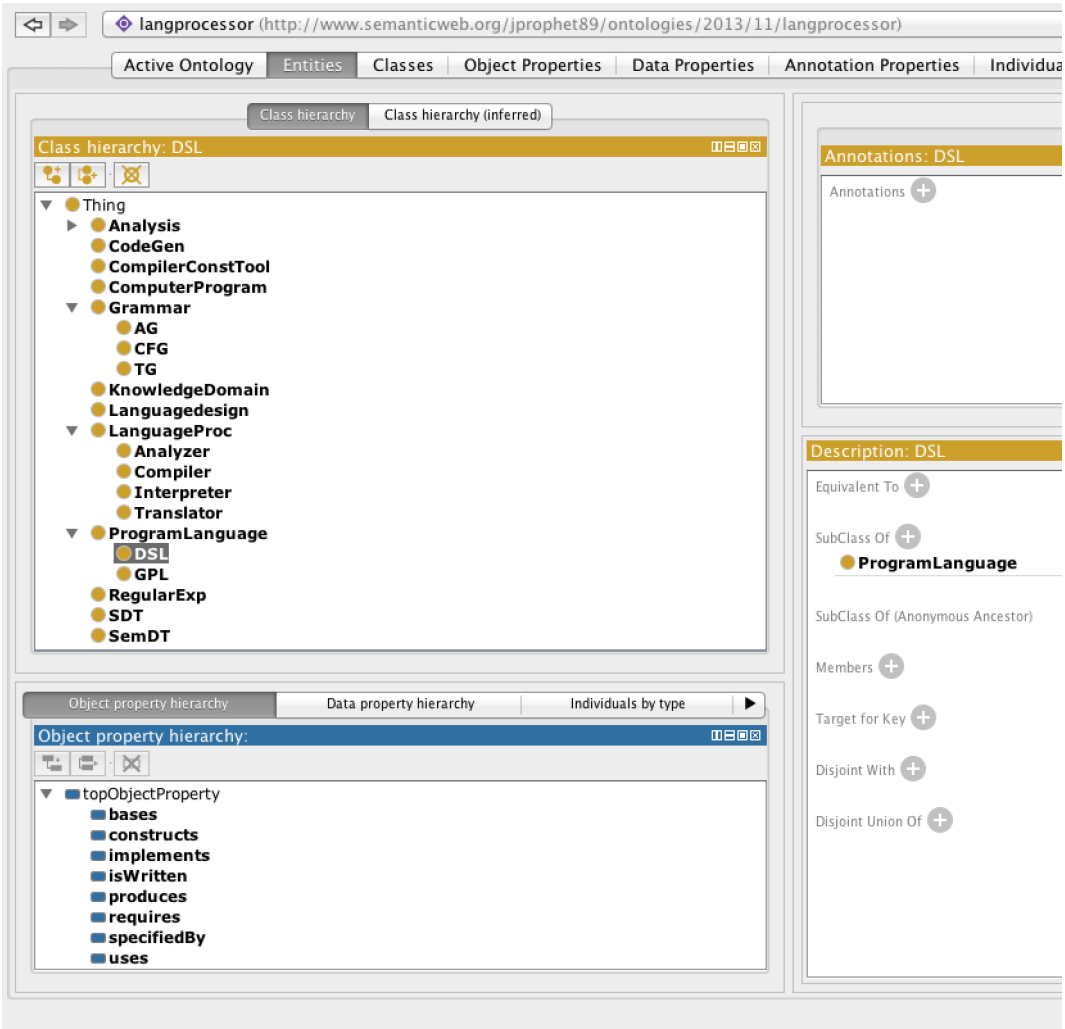

Figure 11.: Protégé screenshot showing the Family ontology generated in OWL/XML

Concluding, this module was created with the objective of generating OWL files from simple ontological descriptions of the domain. It can be used separately from the other components but its objective is to generate input files for the second module of this project, *OWL2DSL*.

5

OWL2DSL MODULE

This section introduces the second module, OWL2DSL, and explains how it is possible to generate a grammar specification and a parser for a DSL to describe elements for the domain. A diagram to sketch the architecture of this module is presented in Figure 12.

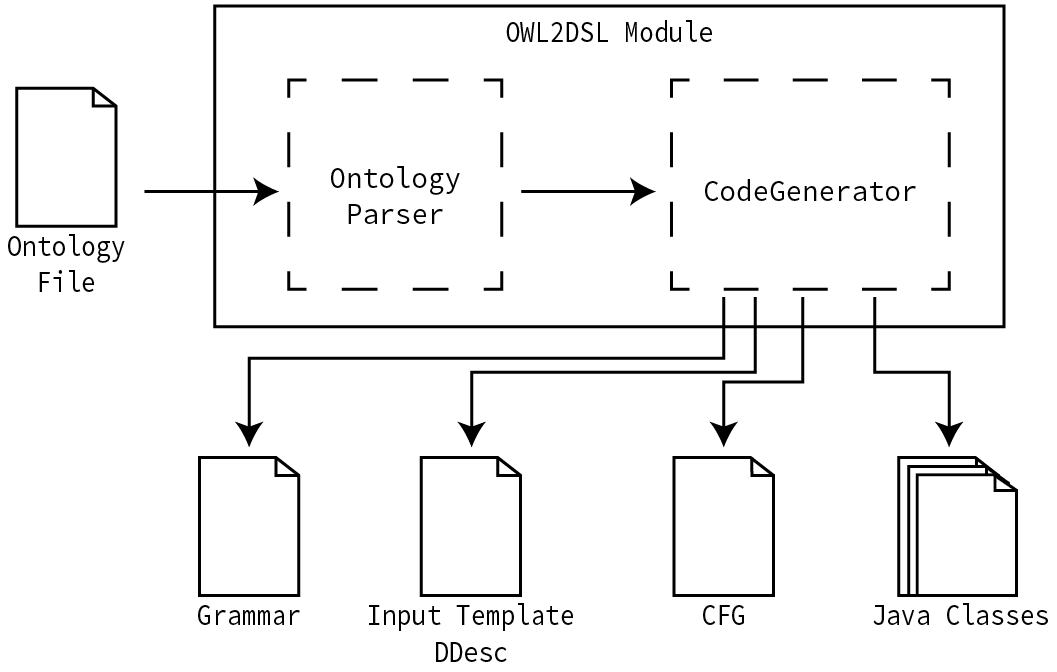

Figure 12.: OWL2DSL Architecture

As we can see in the image above, this processor is composed of two components.

One of them is the Ontology parser. As the studies showed, there are several standard formats to specify an ontology, therefore they must be supported by this module. This component is easy to use because there is no need to specify the input format; this module decides the type and uses the appropriate parser. Figure 13 depicts such choice.

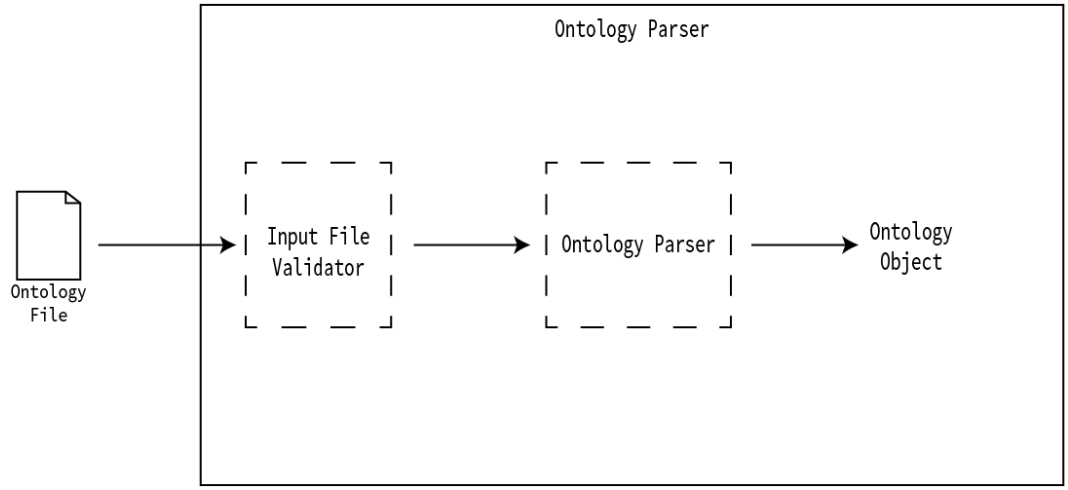

Figure 13.: Ontology Parser schema

No matter the parser activated, the final results that outcomes from this component is an *Ontology Object (OO)*, that contains all the information about the concepts (classes) and relations (hierarchies and properties) extracted from the ontology description contained in the Input file. This information gathered in the *OO* is crucial because it will enable the creation of the desired grammar.

The second module *CodeGenerator*, is a recursive function that analyze the OO and visits all the *Concepts*, *Hierarchical* and non-hierarchical *Relations* connections to generate several files: the desired attribute grammar, a simplified version containing only context free grammar, several Java Classes and a DDesc template.

The first is a grammar with Java code for further processing and the other is the CFG (without the Java code), for better understanding and eventual modification.

This function also generates *DDesc* input template. This functionality allows normal users, that are not familiar with the specified domain and do not read easily a grammar, to know how to write correct sentences.

Finally, the *CodeGenerator*, with the help of a function named “subproduction”, generates a set of classes Java that are helpful in the next module of this project. The Java code generated at this stage, allows for the proper processing of the generated grammar in order to extract information from the source texts and to store it in classes for further process.

This generation process is controlled by a Java class named *CodeGenerator* as schematized in Figure 14.

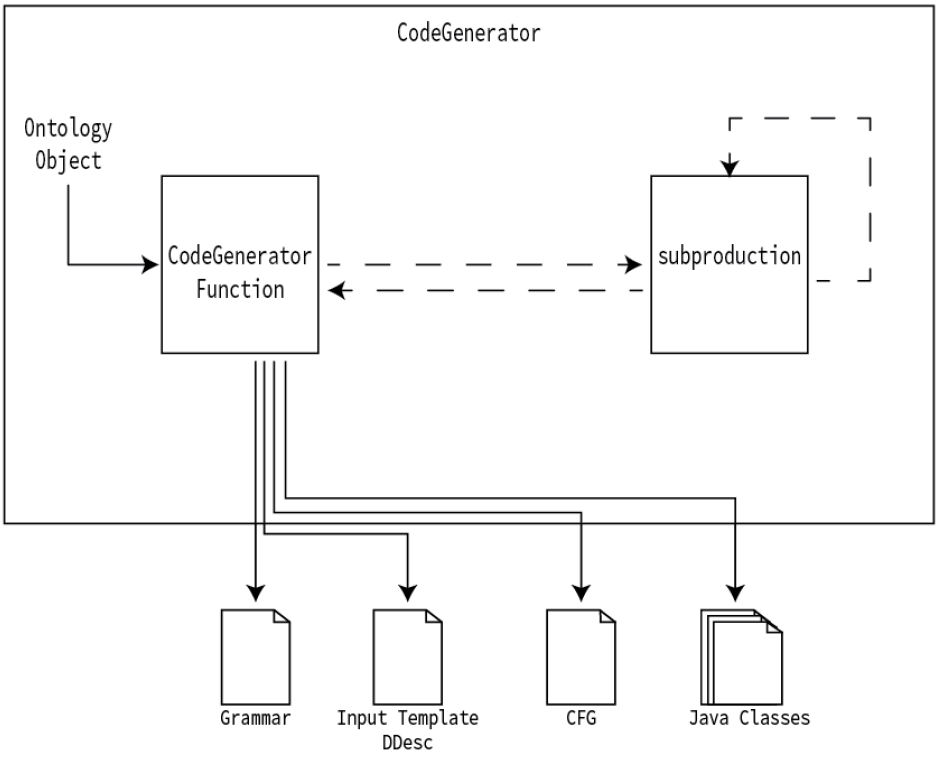

Figure 14.: CodeGenerator schema

Next sub-sections will explain in detail the grammar generation process and the associated rules, the Java classes generation and the Ddesc file template.

5.1 GRAMMAR GENERATION

The main objective of this master thesis was to generate a grammar that represents a certain domain specified by an Ontology written in any OWL standard. Figure 15 represent, a small part from the graph from the language processing ontology, our running example, previously shown in section 4.2 .

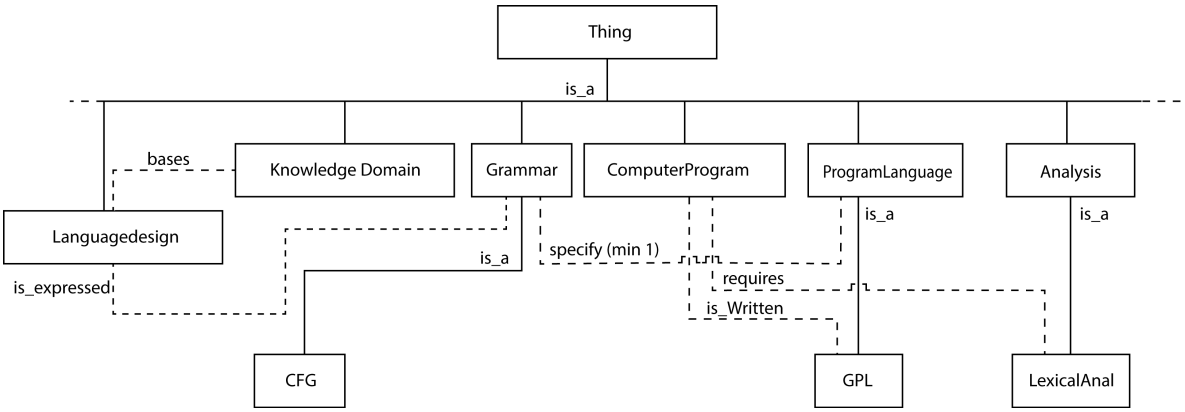

Figure 15.: Ontology Graph

The generation of the grammar is very systematic and obeys a certain rules. The rules are just a few, but without them it was impossible to generate the productions.

The first rule is concerned with the main production. This rule is very simple, it states that if the concept is connected hierarchically to Thing, this concept it will be added to an cycle of super productions. The grammar Axiom on start symbol is named *thing*, it was named based on the ontology schema where the main *Concept* is always "Thing", this is any Concept "is_a" Thing. The first production, with the grammar axiom *thing*, has in its RHS all the *Concepts* that are hierarchically connected to the class "Thing", as alternatives. These alternatives are followed by one iterator to allow zero or more occurrences of each one.

For example if the concept A "is_a" Thing, B "is_a" Thing and C "is_a" A; then in the main production Thing will appear A and B but not C. Listing bellow illustrate how this rule is applied on this first production.

```
Example:: if A is_a Thing and B is_a Thing  and C is_a A

The result::
thing : (A|B)+ ; //C doesn't appear because is not connected hierarchically with
    Thing.
```

Listing 5.1: Thing rule

In our running example, the Hierarchical Concepts connected to "Thing" are twelve. Listing 5.2 shows, with real information, some of those Concepts and the way they are specify on the first production.

```
thing:
  'Thing['
      (
          knowledgedomain|
          languagedesign|
          programlanguage|
          computerprogram|
          ...|
          sdt|
          semdt
      )+ //it is always required at least 1 instance specification
  ']'
;
```

Listing 5.2: Thing production

After the first production, the next step is to decide how to generate the other productions that correspond to the other Concepts.

First was decided that every production would start with an *ID* that specifies the name of the instance of the concept on the LHS. After that we need to address the attributes of that Concept that is being processed. These attributes are optional, this is, it is not mandatory to define these values in the *DDesc* file. The attribute is represented by its the name and its value, that can be a string, int, float or a boolean.

The next big part of the grammar generation process is the hierarchical connections. The rule used was the same that was used on the main production "thing". That means a *Concept* that is a super-class will have each sub-class as a symbol on a RHS alternative. Alternatives are grouped and an iterator operator is added to allow zero or more occurrences of each alternative. Listing bellow show how this particular rule work.

```
Example::
  if B is_a A and C is_a A
    A is a super class and B and C are connected hierarchically to A

The result is::
A : A_ID '[' ( B | C )*  ']' ;
```

Listing 5.3: Hierarchical productions rule

Finally the non-hierarchical connection. This part is different, because it depends on the cardinality of the connections defined between Concepts. The cardinality is given by two parameters presented on the TripleRange class, the min and max. These variables define how the relation will be translated into a grammar rule. The variants are simple to explain. Listing 5.4 shows the cardinalities that

are recognized by our system and how those cardinalities are translate to the grammar. If there is a combination of minimum and maximum the grammar generated will represent that same cardinality in the productions.

The recognized cardinalities are just a few but they are very important for a better grammar generation. Listing 5.4 shows all the possible cardinalities recognized by OWL2GRA.

```
if A bases F then A -> (bases F)*
if A bases min 1 F then A -> (bases F)+
if A bases max 1 F then A -> (bases F)?
if A bases min 2 F then A -> (bases F)  (bases F)+
if A bases max 2 F then A -> (bases F)? (bases F)?
if A bases min 1 max 2 F then A -> (bases F)  (bases F)?
if A bases min 2 max 2 F then A -> (bases F)  (bases F)
```

Listing 5.4: Cardinalities rules

Listing 5.5 will show some fragments of the generated grammar for the same running example to illustrate the application of the rules that were described in the last paragraphs.

```
knowledgedomain:
    'KnowledgeDomain{'
        knowledgedomainID
      (',' '[' (knowledgedomain_bases_languagedesign)* ']' )?
  '}'
;

programlanguage:
    'ProgramLanguage{'
        programlanguageID
      (',' '[' //hierarchical connections
          (gpl | dsl)*
      ']' )?
      (',' '[' //non-hierarchical connections
          (programlanguage_specifiedby_grammar)+ // cardinality min=1
          (programlanguage_specifiedby_regularexp) (
             programlanguage_specifiedby_regularexp)+ // cardinality min=2
      ']' )?
  '}'
;

programlanguage_specifiedby_grammars:
  '{' 'specifiedby' 'grammars' grammarsID '}'
;

programlanguage_specifiedby_regularexp:
```

```
  '{' 'specifiedby' 'regularexp' regularexpID '}'
;

grammarsID:
    STRING
;

regularexpID:
    STRING
;

tg:
    'TG{'
        tgID
      (',' '['
          (tg_implements_sdt)? (tg_implements_sdt)? //cardinality max=2
          (tg_implements_semdt)* //cardinality no min and no max
      ']' )?
  '}'
;
```

Listing 5.5: Generated Productions for different cardinalities

The grammar generated can be read by a Language Engineer and modified or adapted or can be used directly by the last module of *OWL2Gra*. However, if the grammar is modified probably it will not work properly on the *DDesc2OWL* Module.

To complement this generated attribute grammar and in order to be directly processed by the last stage of *OWL2Gra*, a set of Java class will also be generated as described in the next section. These classes store the information extracted from the DDesc input files.

5.2 JAVA CLASS SET GENERATION

The Java classes enables the storage of the information that is gather for the DDesc input file processing. This information is equivalent to Individuals on the OWL terminology. This concrete data will populate that domain with Individuals(or Instances) on the ontology. The Java classes generation is very simple and easy to understand.

The main production of the grammar must save all information from all the hierarchical Classes that are connected to “Thing”. Listing 5.6 bellow will show a generated Java class named “Thing.java” that contains several ArrayLists to store all the Specification.

```
package langprocessor;
import java.util.ArrayList;

public class Thing{
  public ArrayList<KnowledgeDomain> knowledgedomains=new ArrayList<>();
  public ArrayList<Languagedesign> languagedesigns=new ArrayList<>();
  public ArrayList<ProgramLanguage> programlanguages=new ArrayList<>();
  public ArrayList<ComputerProgram> computerprograms=new ArrayList<>();
  public ArrayList<Grammar> grammars=new ArrayList<>();
  public ArrayList<LanguageProc> languageprocs=new ArrayList<>();
  public ArrayList<CompilerConstTool> compilerconsttools=new ArrayList<>();
  public ArrayList<Analysis> analysiss=new ArrayList<>();
  public ArrayList<CodeGen> codegens=new ArrayList<>();
  public ArrayList<RegularExp> regularexps=new ArrayList<>();
  public ArrayList<SDT> sdts=new ArrayList<>();
  public ArrayList<SemDT> semdts=new ArrayList<>();
}
```

Listing 5.6: Generated Thing.java

The other Java Classes are generated based on the *Concept* properties, so each generation is different for each *Concept*. The *Concept* attributes are represented using simple variables. The Hierarchical connections are represented with an ArrayList, as happen for the main class "Thing". This allows the storage of all the hierarchical instances specified on the input. The Relations, non-hierarchical connections, have the same representation with ArrayList as it happens for the hierarchical connections, but instead of saving Concept object it saves the reference for that object.

Listing 5.7 is an example of a generated Java class that contains hierarchical and non-hierarchical connections.

```
package langprocessor;
import java.util.ArrayList;
public class ProgramLanguage{
  public String programlanguageID=null;
      //Hierarchical connections
  public ArrayList<GPL>gpl=new ArrayList<>();
  public ArrayList<DSL>dsl=new ArrayList<>();
      //Non-hierarchical connections
  public ArrayList<String> grammar_specifiedby=new ArrayList<>();
  public ArrayList<String> regularexp_specifiedby=new ArrayList<>();
}
```

Listing 5.7: ProgramLanguage Class generated

The generation of these classes complements the grammar as it was explained in section 5.1 in order to be able to process the DDesc inputs files. These is also another important class generated, the Main. This executable class helps the process of the DDesc Module.

The next section will explain what is a DDesc and how the template is generated.

5.3 DDESC INPUT TEMPLATE

For people not used with programming languages, it could be difficult to create an input file reading the generated grammar. With that in mind, the *CodeGenerator*, while is processing the Ontology to generate the grammar file and the auxiliary Java code, also produces a DDesc input file template.

This template is created with all the possibilities of the grammar. Of course the user does not need to use all the *Concepts* but if needed he can look at the template to understand how it should be specified. This template allows also to understand the cardinality of the relations between *Concepts* and how they should be specified as well.

The listing bellow is an example of a DDesc template automatically generated by the *CodeGenerator* considering again the same running example.

```
Thing[
    KnowledgeDomain{
        "Intance of knowledgedomain ID"
        ,[
            { bases "languagedesignID_reference" } // many possible but need to
                be sequential
        ]
    }
    Languagedesign{
        "Intance of languagedesign ID"
        ,[
            { produces "programlanguageID_reference" } // many possible but need
                to be sequential
        ]
    }
    ProgramLanguage{
        "Intance of programlanguage ID"
        ,[//this instances bellow are not necessary
            GPL{
                "Intance of gpl ID"
            }
            DSL{
                "Intance of dsl ID"
            }
        ]
        ,[
```

```
            { specifiedby "grammarID_reference" } // at least one is required
            { specifiedby "regularexpID_reference" } //  at least two is required
                but more possible
        ]
    }
]
```

Listing 5.8: Generated DDesc Template

As it is possible to see in Listing 5.8, it is very simple to create an input for describing a concrete domain to be processed by the next Module DDesc. This create the possibility to populate the Ontology with many Concept instances just with one simple input file.

6

DDESC2OWL MODULE

DDesc2OWL is the last module of the OWL2Gra. The purpose of this module is to populate the initial ontology that was used to generate the grammar.

The initial ontology is only a specification a rigorous schema of a certain domain. However, if the ontology is populated the domain gains meaning and this allows establishing a network of knowledge that can be used by search engines or used on the Web pages giving them some semantic value. This ontology-based technology is being developing exponentially, because it facilitates the work of the search engines to catalog the information on the pages; this results in better performance and in better quality of the retrieved information.

This Module is very simple; it receives three parameters, that are files, and processes them in different moments.

Figure 16 represents the interface of this module.

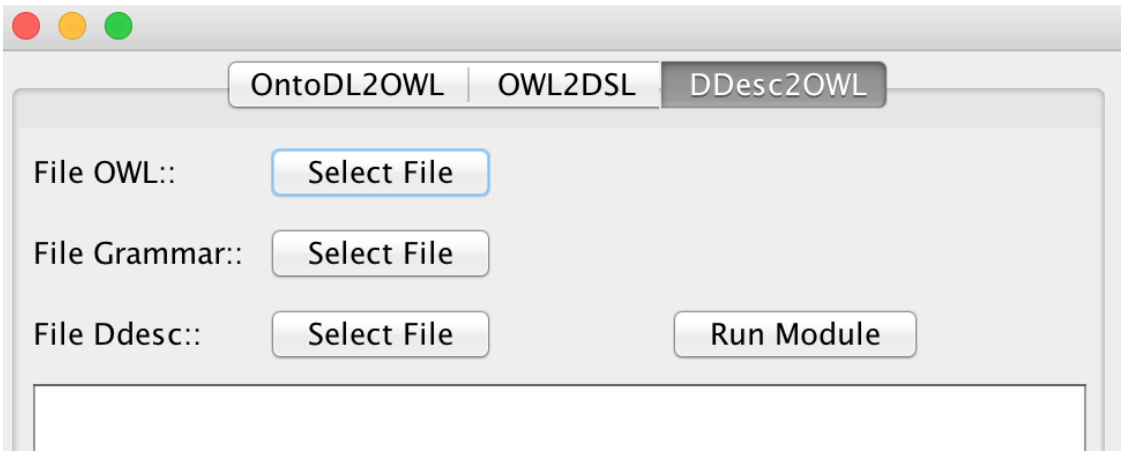

Figure 16.: DDesc2OWL Module Interface

The first important parameter is the path of the grammar file that was generated during the last phase. This grammar will need the Java class set (also generated at the previous stage).

The second file is the DDesc input file. This file describes all the instances that will create the individuals on the ontology.

Finally, the last file is the original ontology that was used for the creation of the grammar by the OWL2DSL Module. This file will be processed again to rebuild the ontology in order to create the populated version.

Figure 17 represents a schema of this module that will be described with full details in the next sections.

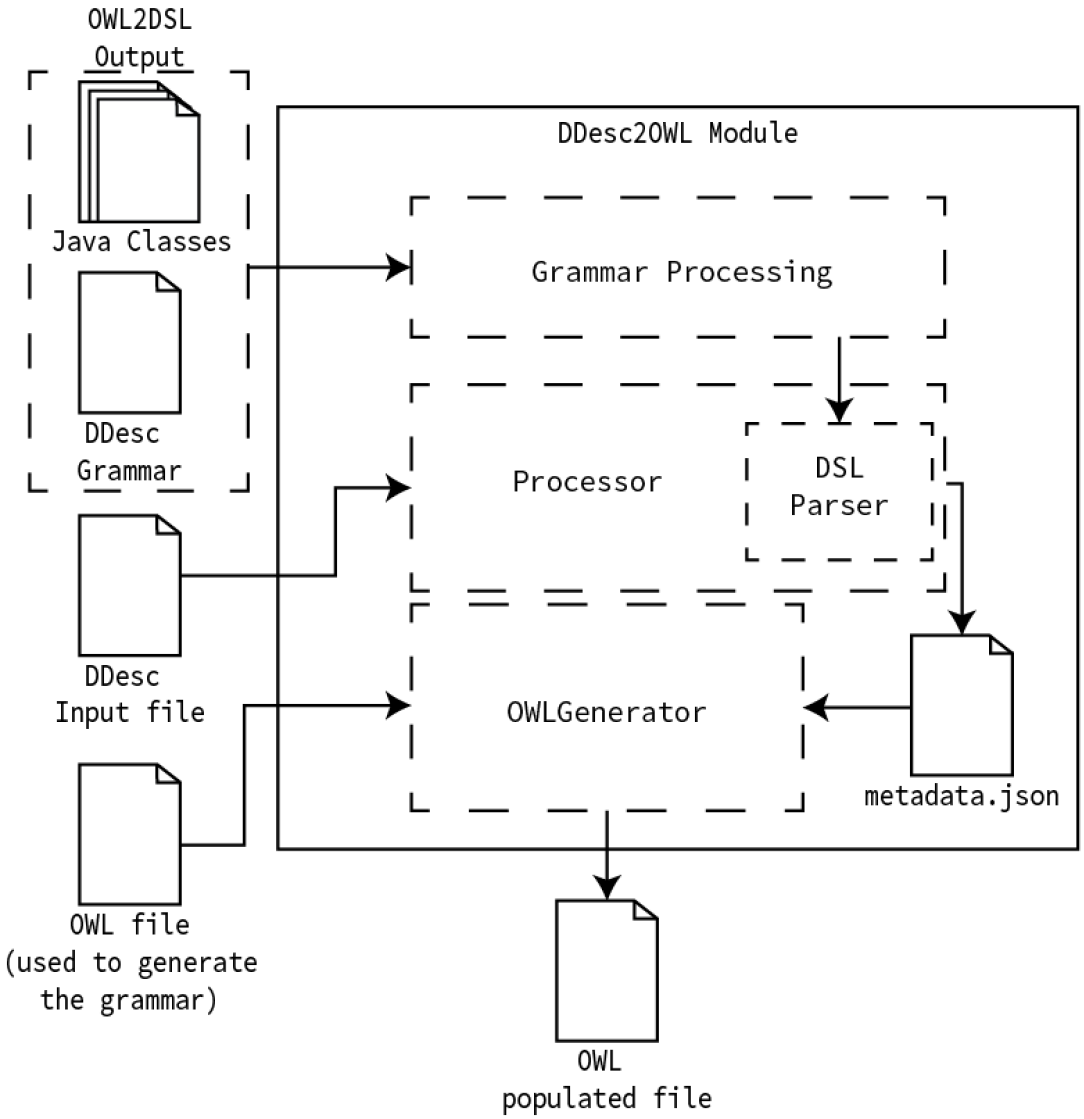

Figure 17.: DDesc2OWL Module schema

6.1 GRAMMAR DDESCG PROCESSOR

In order to use the grammar generated at the second phase (by OWL2DSL module), it is required to use a compiler generator like AnTLR to generate the Parser and Lexer files that will be used to process the DDesc File.

This *Jar file*, named "antlr-3.5.1-complete.jar", is included on the OWL2Gra program. The Parser and Lexer are generated using a single system call.

Listing 6.1 describes how this grammar processor works; it makes a process involving the compiler generator chosen (AntLR) with the grammar file as argument. If there is a problem with the Parser and Lexer generation the *TextArea* on the module interface will show the error.

```
Process p=Runtime.getRuntime().exec("java -jar antlr-3.5.1-complete.jar -Dfile.
    encoding=utf-8 "+grammarfile.getAbsolutePath());
p.waitFor(); //wait that the generation is completed.
if(p.exitValue()==0){
    DataInputStream dis = new DataInputStream(p.getErrorStream());
    String line = "";
    while ( (line = dis.readLine()) != null)
    {
        jTextArea1.setText(jTextArea1.getText()+"\n"+line);
    }
    dis.close();
}else{
    DataInputStream dis = new DataInputStream(p.getErrorStream());
    String line = "";
    while ( (line = dis.readLine()) != null){
        jTextArea1.setText(jTextArea1.getText()+"\n"+line);
    }
    dis.close();
}
```

Listing 6.1: Grammar Pre-Processing

The DDesc Processor, that is generated by this process, is the important part of this module, because it is responsible for recognizing and gathering all the information that is represented in the DDesc input file. Without this Processor, it was not possible to retrieve the population that is in the DDesc file.

The DDesc Processor (generated at this stage) will be explained in the next section.

6.2 DSL(DDESC) PROCESSOR

In this part, it is crucial that all the information is collected properly in the right places so that this process must run perfectly.

Several components are necessary to execute this phase: all the Java classes that were generated by the last Module OWL2DSL and the Parser and Lexer that were generated in the previous stage. In addiction, this process uses the file Main.java that was created by the last Module.

Listing 6.2 shows how the DDesc2OWL handles this process. However, if any error occurs, exit value different than zero, it will be printed in the *TextArea* of the Module interface.

```
p=Runtime.getRuntime().exec("javac -cp antlr-3.5.1-complete.jar:commons-io-2.4.
    jar:\$CLASSPATH:dom4j-2.0.0-ALPHA-2.jar:json-simple-1.1.1.jar:. -encoding utf8
     Main.java");
if(p.exitValue()==0){
    DataInputStream dis = new DataInputStream(p.getErrorStream());
    String line = "";
    while ( (line = dis.readLine()) != null)
    {
        jTextArea1.setText(jTextArea1.getText()+"\n"+line);
    }
    dis.close();
}else{
    DataInputStream dis = new DataInputStream(p.getErrorStream());
    String line = "";
    while ( (line = dis.readLine()) != null){
        jTextArea1.setText(jTextArea1.getText()+"\n"+line);
    }
    dis.close();
}
```

Listing 6.2: Grammar Pre-Processing

If DDesc input is not properly written, the errors will be detected and displayed in the *TextArea* of DDesc2OWL interface.

Figure 18 bellow demonstrates the processing of an example in order to demonstrate the type of the alerts messages that are issued to the user. This allows the user to correct the DDesc input easily and quickly.

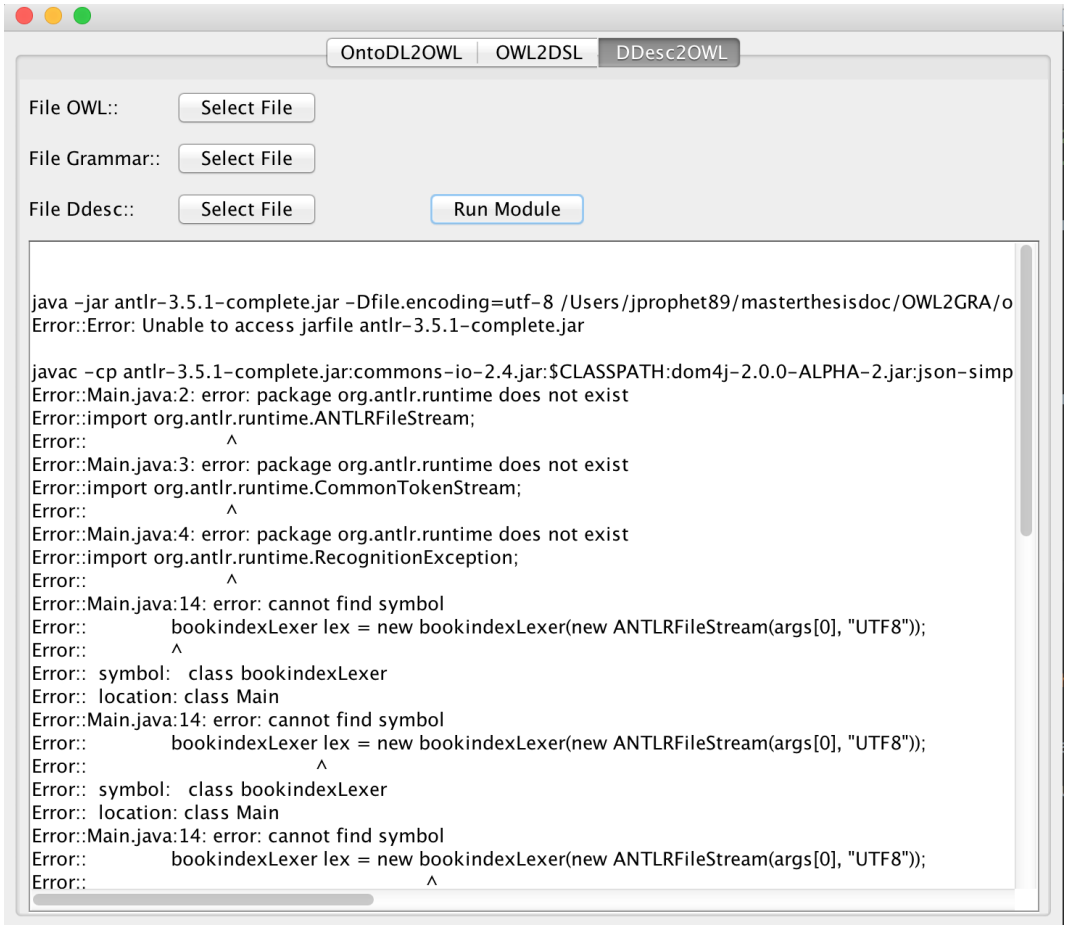

Figure 18.: DDesc error alert example

If the process is completed and no errors are detected none of these error messages will appear and the user can understand that everything is correctly processed. In this case, when the processing is completed with success, the program will create an ontology populated with extracted data.

The result of this grammar process is a file with the Meta information of the instances described in the DDesc input file. This file is used by the OWL Generator, to generate the OWL. This generation uses a certain set of rules for the proper creation of OWL individuals.

This Meta information is stored in a Json file. The file contains an array of objects that represents the Individuals of the ontology. These objects have the same specification; first they contain two pairs that are mandatory, “name” and “type”, these pairs represent the name of each individual and the ontology class where belongs to. In addition, there are another two pairs, not mandatory, that represent information that can be or not be present in the individual specification contained in the DDesc file. These two pairs are the “dataprop” and the “objprop”; both have the same structure.

The “dataprop”, as the name shows, is used to store the Data Properties of the Individual. In the array objects, it is specified what is the Data property and its value. As referred above, this array may be empty because not every class have Data Properties.

The “objprop” is very similar to the “dataprop”. This time the array is used to store the non-hierarchical relation between Individuals. The objects in the array store the same type of information that the Data properties, but in this case it stores the name of the Object properties and the Individual that it is connected.

Listing 6.3 shows an example of the Json that is being generated for the processed grammar.

```
{
    "instancias":[
        {
            "name":"Domain_Knowledge_individual_name1",
            "type":"KnowledgeDomain",
            "objprop":[
                {
                    "data":"languagedesignID",
                    "prop":"bases"
                }
            ]
        },{
            "name":"Domain_Knowledge_individual_name2",
            "type":"KnowledgeDomain",
            "objprop":[
                {
                    "data":"languagedesignID",
                    "prop":"bases"
                }
            ]
        }
    ]
}
```

Listing 6.3: Metadata File generated

The grammar processor is just a middle processor for this module. Next section explains the final part of this module that generates the OWL file with the individuals.

6.3 OWL GENERATOR

This component of the third module, gathers all the information from the instances described in the input test and add that information to the original ontology file.

The first part of this process is to get the initial OWL file so that no reference or something is missing in the final ontology file.

The same OWL API that is used to parse the ontology on the *OWL2DSL* Module, now its used to add the information of the individuals on the final ontology.

To be able to generate these individuals the module should have access to the generated file with the Meta information that was described in the section above.

Listing bellow shows this first verification on the module.

```
File metajson=new File("metadata.json");
if(!metajson.exists()){
    output+="\nERROR::Metadata.json was not found the process was not completed!!
        ";
    return output;
}
```

Listing 6.4: Validation of the “metadata.json”

After the verification, the module is ready to create the individuals with the help of the OWL API. This API allows the module to add individuals without having to rewrite the ontology from the scratch.

In order to create the sequence of individuals the “metadata.json” is processed and for each object inside the same process occurs.

First is extracted the “name” and the “type” from the object that is processed. This information will help on the declaration of the Individual.

The Listing 6.5 explains how this information is used by the OWL API to generate the Individual.

```
JSONObject obj=i.next();//get the object that will be processed
//Individual and Class type generation
OWLNamedIndividual name =df.getOWLNamedIndividual(IRI.create(obj.get("name").
    toString()));
OWLClass type = df.getOWLClass(IRI.create(ontology.getOntologyID().getOntologyIRI
    ()+"#"+obj.get("type").toString()));
OWLClassAssertionAxiom classAssertion = df.getOWLClassAssertionAxiom(type, name);
manager.addAxiom(ontology,classAssertion);
```

Listing 6.5: Declaration of the individual

In addiction, it was explained that this object that contains the individual information has two pairs that are not mandatory, the “dataprop” and the “objprop”.

The first to be processed is the “dataprop” array. This array contains the object that specifies all the Data properties that need to be add to the individual that was declare in the example 6.5. To be able to do this the object contains two important keys that are “prop”, specifies the name of the Data Property, and the “data” that specify the value of that property.

The Listing bellow shows an example of how this property is added to the individual.

```
if(obj.containsKey("dataprop")){
    JSONArray object=(JSONArray)obj.get("dataprop");
    for(Iterator<JSONObject> o = object.iterator();o.hasNext();){
        JSONObject otemp=o.next();
        OWLDataProperty prop = df.getOWLDataProperty(IRI.create(ontology.
            getOntologyID().getOntologyIRI() + "#" + otemp.get("prop").toString())
            );
        OWLDataPropertyAssertionAxiom dataPropertyAssertionAxiom = df.
            getOWLDataPropertyAssertionAxiom(prop, name, otemp.get("data").
            toString());
        manager.addAxiom(ontology,dataPropertyAssertionAxiom);
    }
}
```

Listing 6.6: Data Property specification example

The final processing array is the "objprop'that represents the Object Properties. The non-hierarchical connection between individuals is specified by this array as it was explained before. The objects in the array have only two keys, like the objects presented on the Data Properties. However this time the "prop" key saves the value of the Object Property that will be used and the "data" represents the Individual that is connect by the property to the individual.

Listing 6.7 represent the creation process of this Object Properties.

```
//ObjectProperties generation
if(obj.containsKey("objprop")){
    JSONArray object=(JSONArray)obj.get("objprop");
    for(Iterator<JSONObject> o = object.iterator();o.hasNext();){
        JSONObject otemp=o.next();
        OWLObjectProperty prop = df.getOWLObjectProperty(IRI.create(ontology.
            getOntologyID().getOntologyIRI()+"#"+otemp.get("prop").toString()));
        OWLNamedIndividual connected_indiv =df.getOWLNamedIndividual(IRI.create(
            otemp.get("data").toString()));
        OWLObjectPropertyAssertionAxiom propertyAssertion = df.
            getOWLObjectPropertyAssertionAxiom(prop,name,connected_indiv);
        manager.addAxiom(ontology,propertyAssertion);
    }
}
```

Listing 6.7: Object Property specification example

The result of this module is the population of the original ontology. This allows to use the outcome to reach other objectives rather than just describing the domain. As was refereed before a populated

ontology can be used for several cases, like web-semantic that allows better querying results from the web search engines. An offline use is also possible, it can be used as a small database and can use SparQL querying to search the information.

Listing 6.8 shows a simple example of a DDesc input file that will create two individuals on the ontology through the OWL Generator that was described.

```
Thing[
  KnowledgeDomain{
    "KnowledgeDomain" //name of the individual
    ,[  { bases languagedesign "Java" } ]//connection to other individuals
  }
  Languagedesign{
    "Java"
    ,[  { produces programlanguage "Java_Language" } ]
  }
]
```

Listing 6.8: DDesc input example

Figure 19 represents the ontology with the generated individuals by the Module DDesc2OWL from the previous example of the DDesc input, opened with Protégé system.

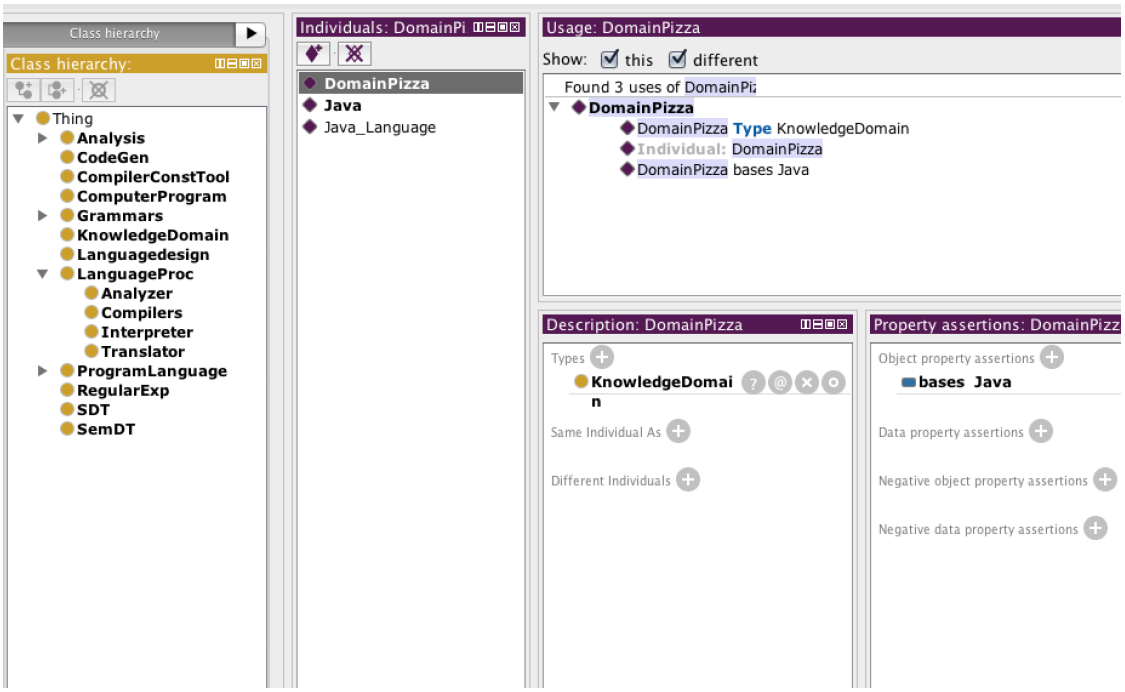

Figure 19.: Final Ontology with individuals loaded into Protégé

6.3. OWL Generator

This module was not included in the original objective but during the development it was possible to understand the possibility to create it easily and it become clear how useful it could be to populate an entire ontology with only one simple input.

7

CONCLUSION

The masterwork that was described and proposed along this thesis report was challenging and has potential, because these two technologies addressed in this master thesis (ontologies and grammars) can work perfectly together like it was proved.

The work reported had a slow start with the collection and study of the most important state of the art technologies used in the two referred areas but it speed up when the development phase started. In the beginning of this master thesis, the objective was only to prove the possibility of converting Ontology to a Domain Specific Language generating its grammar. However, with the need of Ontologies to further test the generation during the development, a new objective was added to the project: that was to allow creating an Ontology through a Domain Specific Language. To realize this objective, a new module was added: Onto2OWL.

This new module, Onto2OWL, allows the creation of a simple ontology based on simple sentences that describes the domain and all its properties in terms of relations between concepts. This proves that not only is possible to create an ontology based on a simple description file, such as the one written in OntoDL (Ontology Description Language), but also proves to be very useful for learning what is ontology and how it can be specified.

The development of this first module was very useful for the main phase of this work that was to transform ontology into an attribute grammar for a Domain Specific Language. The progress of the work has shown that it is necessary to add some annotations to the ontology in order to generate a full attribute grammar.

The Domain and the Range of the Object Properties are not specified in a proper way that allows the Code Generator, see chapter 5, and process correctly them. The problem was found when different Concepts are connected to different Ranges with the same property; in that case, the Code generator will assume that all the Domains have all the Ranges. This assumption leads to the generation of a grammar with errors. This problem was easily overcome with OWL Annotations that tell the parser, what are the proper ranges for each Domain. The annotations are very simple to process and avoid the errors on the attribute grammar. If the grammar does not present annotations, the relations will be ignored and an error message issued.

With the addiction of cardinality to the properties, the Code Generator is able to generate a more correct and complete grammar.

After the development of this module, when the idea of generating a grammar from an Ontology proven feasible, the possibility of using the same generated grammar to populate the Ontology was proposed and some modifications were made to the Code Generator that was responsible to generate the grammar.

The Code Generator was changed in order to generate two grammars from the Ontology. Both grammars describe the same syntax; have the same productions and syntactic sugar. However, one of them has attributes and some embedded code that allows the storage of information for further processing.

For a user that is not familiar with attribute grammars it is not easy to produce a correct input. To overcome this problem, the Code Generator suffers other change in order to generate a template for the domain description. This allows users without experience on the Domain Specific Language, to work with this tool and obtain the results they are expecting.

This template is useful to produce the input description that will be feed to the third module that was created with the intention of populating the Ontology (that was used to create the attribute grammar) with the data that can be extracted from a Domain Description File.

This is a smart way to populate on Ontology because it reduces time and effort on this task. The template supports the creation of many individuals and establishes the individuals connections.

To develop this module we faced among some problems with the proper generation of the individuals, but all the problems were resolved easily. This module is proving that the inverse path of the one followed OWL2DSL is also possible, and that a DSL can generate ontology with individuals that can be used for many purposes.

The possibility given by the command line support is important because it makes the OWL2Gra a tool that can be used by other system and run on other systems.

Finally, it is possible to state that the outcomes of this master thesis work accomplished all the initial objectives that were proposed and the other objectives that were created along the way. These Modules can be used separately for different purposes. This allows a customizable use of this tool that proves that is possible to generate a semantic sugared attribute grammar from Ontology.

BIBLIOGRAPHY

Ines Ceh, Matej Crepinsek, Tomaz Kosar, and Marjan Mernik. Ontology driven development of domain-specific languages, 2011.

D.Jin. Ontological adaptive integration of reverse engineering tools, 2004.

Rosario Girardi. The hermes project advances and challenges, 2010.

Fausto Giunchiglia, Biswanath Dutta, and Vincenzo Maltese. Faceted lightweight ontologies, 2009.

S. Grimm. Knowledge representation and ontologies, in m. gaber. (eds.) scientific data mining and knowledge discovery: Principles and foundations, 2010.

Tomaz Kosar, Pablo E. Martínez López, Pablo A. Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language, 2008.

Nuno Ernesto Salgado Oliveira. Improving program comprehension tools for domain specific languages, 2010.

Jeff Gray Robert Tairas, Marjan Mernik. Using ontologies in the domain analysis of domain-specific languages, 2009.

Robert Tairas, Marjan Mernik, and Jeff Gray. Using ontologies in the domain analysis of domain-specific languages, 2009.

Arie van Deursen and Paul Klint. Domain-specific language design requires feature descriptions, 2001.

A

CASE STUDY 1 - BOOK INDEX

This case study reflects the daily work that is found on a normal day at some book store, the book index. This is a very simple example that show how easy is to describe a domain like this and demonstrate how OWL2Gra processes all the information and manage the cardinality restrictions in the relations between Individuals.

A.1 BOOK INDEX ONTODL

This DSL show how easy is to define the things that represent the Book.

The listing bellow show the OntoDL.

```
Ontology{
  Concepts[
    {Book},
    {
      Page,
      Attributes[
        {text string},
        {number int}

      ]
    },
    {Title},
    {SpecialTerm}
  ]
  Hierarchies[
  ]
  Relations[
    {has},
    {contains}
  ]
  Links[
    {Book has Page},
    {Book has min 1 Title},
```

```
    {Page contains SpecialTerm}
  ]
}
```

Listing A.1: Book Index OntoDL

A.2 BOOK INDEX GENERATED OWL

The listing bellow show the OWL that has generated from Onto2OWL Module.

```
<?xml version="1.0"?>
<Ontology xmlns="http://www.w3.org/2002/07/owl#"
     xml:base="http://example.com/onto2owl/Bookindex/"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:xml="http://www.w3.org/XML/1998/namespace"
     ontologyIRI="http://example.com/onto2owl/Bookindex/">
    <Prefix name="rdf" IRI="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
    {...}
    <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
    <Declaration>
        <Class IRI="#Book"/>
    </Declaration>
    {...}
    <Declaration>
        <ObjectProperty IRI="#contains"/>
    </Declaration>
    {...}
    <Declaration>
        <DataProperty IRI="#number"/>
    </Declaration>
    {...}
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#contains"/>
        <Class IRI="#Page"/>
    </ObjectPropertyDomain>
    {...}
    <ObjectPropertyRange>
        <Annotation>
            <AnnotationProperty abbreviatedIRI="owl:backwardCompatibleWith"/>
            <IRI>#Page</IRI>
        </Annotation>
        <ObjectProperty IRI="#contains"/>
        <Class IRI="#SpecialTerm"/>
```

```
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <Annotation>
            <AnnotationProperty abbreviatedIRI="owl:backwardCompatibleWith"/>
            <IRI>#Book</IRI>
        </Annotation>
        <ObjectProperty IRI="#has"/>
        <Class IRI="#Page"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <Annotation>
            <AnnotationProperty abbreviatedIRI="owl:backwardCompatibleWith"/>
            <IRI>#Book</IRI>
        </Annotation>
        <ObjectProperty IRI="#has"/>
        <ObjectMinCardinality cardinality="1">
            <ObjectProperty IRI="#has"/>
            <Class IRI="#Title"/>
        </ObjectMinCardinality>
    </ObjectPropertyRange>
    <DataPropertyDomain>
        <DataProperty IRI="#number"/>
        <Class IRI="#Page"/>
    </DataPropertyDomain>
    {...}
    <DataPropertyRange>
        <DataProperty IRI="#text"/>
        <Datatype IRI="string"/>
    </DataPropertyRange>
</Ontology>
```

Listing A.2: Book index generated OWL

A.3 BOOK INDEX GENERATED GRAMMAR

Listing bellow shows the resulting grammar of OWL2DSL from the OWL presented on listing A.2.

```
grammar Bookindex;
thing:
  'Thing['(page|title|specialterm|book)+ ']'
;


page: 'Page{'pageID ('number' number)?('text' text)?
  (',' '[' (page_contains_specialterm)* ']' )?'}'
```

```
;

pageID :
   STRING
;
number:
   INT
;
text:
   STRING
;
page_contains_specialterm:
  '{' 'contains' 'specialterm' specialtermID'}';

title:  'Title{'titleID '}'
;

titleID :
   STRING
;

specialterm:  'SpecialTerm{'specialtermID '}'
;

specialtermID :
   STRING
;

book: 'Book{'bookID
  (',' '[' (book_has_page)*(book_has_title)+ ']' )?'}'
;

bookID :
   STRING
;
book_has_page:
  '{' 'has' 'page' pageID'}';
book_has_title:
  '{' 'has' 'title' titleID'}';
```

Listing A.3: Book index Grammar generated

A.4 BOOK INDEX DDESC INPUT

Listing A.4 shows an possible input for the last module in DDesc terminology.

```
Thing[
  Page{
    "GOT_page1"
    text "Concent of the page"
  }
  Book{  "GOT_Book"
    ,[
      { has page "GOT_page1" }
      { has title "Game of Thrones" }
    ]
  }
  Title{
    "Game of Thrones"
  }
]
```

Listing A.4: Book index DDesc input example

A.5 BOOK INDEX DDESC2OWL RESULT

In order to complete this Figure 20 represent the final result on DDesc2OWL module, the original ontology with individuals.

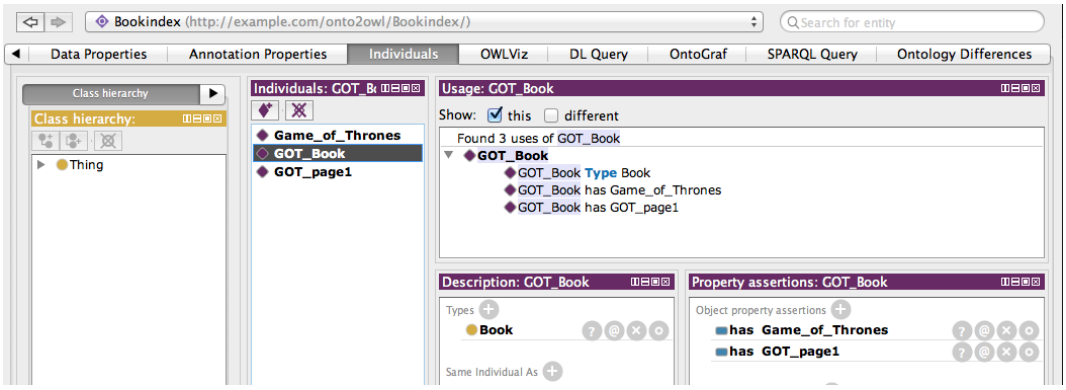

Figure 20.: DDesc2OWL final result Case Study 1 opened in Protégé

B

CASE STUDY 2 - LAUNDRY PROCESS

The case study is about a very different domain, the daily process of a Laundry.

In this case study is possible to demonstrate properties like multiple attributes and also the cardinality restrictions on Links.

B.1 LAUNDRY ONTODL

Listing B.1 shows a clear example of multiple attributes on Concept “Type” and also cardinality restrictions on the last two links.

```
Ontology{
  Concepts[
    {Laundry},
    {Order},
    {Client},
    {Bag},
    {Item},
    {
      Type,
      Attributes[
            {classes string},
            {tinge string},
                {material string}
            ]
    },
    {Quantity}
  ]
  Hierarchies[
  ]
  Relations[
        {has},
        {owns},
        {receives},
        {contains}
```

```
  ]
  Links[
        {Laundry receives Order},
        {Client owns Order},
        {Order contains Bag},
        {Bag contains Item},
        {Item has max 1 Type},
        {Item has max 1 Quantity}
  ]
}
```

Listing B.1: Laundry OntoDL

B.2 LAUNDRY GENERATED OWL

Listing B.2 show the OWL that has generated from Onto2OWL Module.

```
<?xml version="1.0"?>
<Ontology xmlns="http://www.w3.org/2002/07/owl#"
     xml:base="http://example.com/onto2owl/Laundry/"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:xml="http://www.w3.org/XML/1998/namespace"
     ontologyIRI="http://example.com/onto2owl/Laundry/">
    <Prefix name="rdf" IRI="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
    {...}
    <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
    <Declaration>
        <Class IRI="#Bag"/>
    </Declaration>
    <Declaration>
        <Class IRI="#Client"/>
    </Declaration>
    <Declaration>
        <Class IRI="#Item"/>
    </Declaration>
    <Declaration>
        <Class IRI="#Laundry"/>
    </Declaration>
    {...}
    <Declaration>
        <ObjectProperty IRI="#contains"/>
    </Declaration>
    {...}
```

```
    <Declaration>
        <ObjectProperty IRI="#receives"/>
    </Declaration>
    {...}
    <Declaration>
        <DataProperty IRI="#tinge"/>
    </Declaration>
    {...}
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#receives"/>
        <Class IRI="#Laundry"/>
    </ObjectPropertyDomain>
    <ObjectPropertyRange>
        <Annotation>
            <AnnotationProperty abbreviatedIRI="owl:backwardCompatibleWith"/>
            <IRI>#Order</IRI>
        </Annotation>
        <ObjectProperty IRI="#contains"/>
        <Class IRI="#Bag"/>
    </ObjectPropertyRange>
    {...}
    <ObjectPropertyRange>
        <Annotation>
            <AnnotationProperty abbreviatedIRI="owl:backwardCompatibleWith"/>
            <IRI>#Item</IRI>
        </Annotation>
        <ObjectProperty IRI="#has"/>
        <ObjectMaxCardinality cardinality="1">
            <ObjectProperty IRI="#has"/>
            <Class IRI="#Type"/>
        </ObjectMaxCardinality>
    </ObjectPropertyRange>
    <DataPropertyDomain>
        <DataProperty IRI="#material"/>
        <Class IRI="#Type"/>
    </DataPropertyDomain>
    {...}
    <DataPropertyRange>
        <DataProperty IRI="#tinge"/>
        <Datatype IRI="string"/>
    </DataPropertyRange>
</Ontology>
```

Listing B.2: Laundry generated OWL

B.3 LAUNDRY GENERATED GRAMMAR

Listing B.3 represent one of the outcomes from OWL2DSL. In this grammar its possible to see clearly in the “type” production how the Data properties(attributes on OntoDL) are processed and generated.

```
grammar Laundry;
thing:
  'Thing['(type|laundry|bag|client|item|order|quantity)+ ']'
;


type: 'Type{'typeID ('tinge' tinge)?('material' material)?('classes' classes)?'
  }'
;

typeID :
   STRING
;
tinge:
   STRING
;
material:
   STRING
;
classes:
   STRING
;

laundry:  'Laundry{'laundryID
  (',' '[' (laundry_receives_order)* ']' )?'}'
;

laundryID :
   STRING
;
laundry_receives_order:
  '{' 'receives' 'order' orderID'}';

bag:  'Bag{'bagID
  (',' '[' (bag_contains_item)* ']' )?'}'
;

bagID :
   STRING
;
bag_contains_item:
```

```
  '{' 'contains' 'item' itemID'}';

client: 'Client{'clientID
  (',' '[' (client_owns_order)* ']' )?'}'
;

clientID :
   STRING
;
client_owns_order:
  '{' 'owns' 'order' orderID'}';

item: 'Item{'itemID
  (',' '[' (item_has_quantity)?(item_has_type)? ']' )?'}'
;

itemID :
   STRING
;
item_has_quantity:
  '{' 'has' 'quantity' quantityID'}';
item_has_type:
  '{' 'has' 'type' typeID'}';

order:  'Order{'orderID
  (',' '[' (order_contains_bag)* ']' )?'}'
;

orderID :
   STRING
;
order_contains_bag:
  '{' 'contains' 'bag' bagID'}';

quantity: 'Quantity{'quantityID '}'
;

quantityID :
   STRING
;
```

Listing B.3: Laundry generated Grammar

B.4 LAUNDRY DDESC INPUT

In this DDesc input is possible to understand the terminology in the “Type” specification, that is used to describe the Data Properties.

Listing B.4 shows how easy is to write several individuals and even to link Individuals to other without specify them like “bag_2”.

```
Thing[
  Client{
    "Client_1"
    ,[
      { owns order "order_2" }
      { owns order "order_4" }
    ]
  }
  Client{
    "Client_2"
    ,[
      { owns order "order_1" }
      { owns order "order_3" }
    ]
  }
  Item{
    "item_1"
    ,[
      { has type "type_1" }
      { has quantity "quantity_private" }
    ]
  }
  Type{
    "type_1"
    tinge "colour_full"
    classes "colour_cloth"
    material "not fibers"
  }
  Laundry{
    "Laundry_name"
    ,[
      { receives order "order_1" }
      { receives order "order_2" }
      { receives order "order_3" }
      { receives order "order_4" }
    ]
  }
  Bag{
    "bag_1"
```

```
    ,[
      { contains item "item_1" }
      { contains item "item_2" }
    ]
  }
  Quantity{
    "quantity_private"
  }
  Order{   "order_1"
    ,[
      { contains bag "bag_1" }
    ]
  }
  Order{   "order_2"
    ,[
      { contains bag "bag_2" }
    ]
  }
  Order{   "order_3"
    ,[
      { contains bag "bag_1" }
    ]
  }
  Order{   "order_4"
    ,[
      { contains bag "bag_2" }
    ]
  }
]
```

Listing B.4: Laundry Process DDesc input

B.5 LAUNDRY PROCESS DDESC2OWL RESULT

In the example above it was shown that one of the individuals was not described on the Ddesc input, but once the Reasoner starts to process the ontology it associate the individual “bag_2” with the class “Bag”.

Figure 21 represent the final result of the same input.

B.5. Laundry Process DDesc2OWL Result

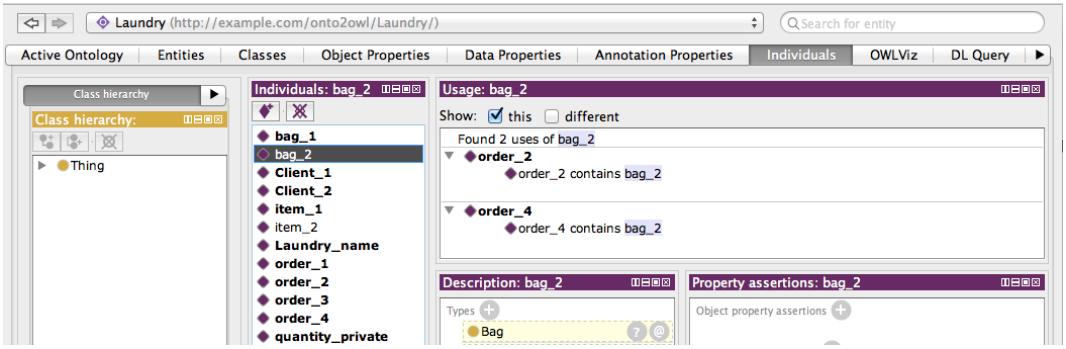

Figure 21.: DDesc2OWL final result on Case Study 2 opened in Protégé

Figure 21 demonstrate even when the user forget to describe a concept in the DDesc input,the module will create that individual without a type specify. However using the Reasoner, the individuals will be classified by the Reasoner logic processor. In this case “bag_2” it will be associated with the Class “Bag”.

C

CASE STUDY 3 - LANGUAGE PROCESSING DOMAIN

The particularity of this case study is to demonstrate that the OWL2Gra also supports big and complex domains.

There is two parts that are important to retain from this case study; the complexity of the domain and also the way that hierarchies are handle in all the OWL2Gra modules.

C.1 LANGUAGE PROCESSING ONTODL

This first section demonstrate an big and complex ontology being described using a simple language as OntoDL.

Listing C.1 shows how easy is to describe a complex domain using a simple notation and also how hierarchies are describe in OntoDL terminology.

```
Ontology{
  Concepts[
    {KnowledgeDomain},
    {Languagedesign},
    {GPL},
    {DSL},
    {ProgramLanguage},
    {ComputerProgram},
    {Grammars},
    {LanguageProc},
    {CompilerConstTool},
    {Interpreter},
    {Analyzer},
    {Translator},
    {Compilers},
    {Analysis},
    {LexicalAnal},
    {SyntacticAnal},
    {SemanticAnal},
    {CodeGen},
```

```
    {RegularExp},
    {CFG},
    {TG},
    {AG},
    {SDT},
    {SemDT}
  ]
  Hierarchies[
    {ProgramLanguage > GPL},
    {ProgramLanguage > DSL},
    {LanguageProc > Interpreter},
    {LanguageProc > Compilers},
    {LanguageProc > Analyzer},
    {LanguageProc > Translator},
    {Grammars > CFG},
    {Grammars > TG},
    {Grammars > AG},
    {Analysis > LexicalAnal},
    {Analysis > SyntacticAnal},
    {Analysis > SemanticAnal}
  ]
  Relations[
    {bases},
    {produces},
    {isWritten},
    {requires},
    {uses},
    {implements},
    {specifiedBy},
    {is_expressed},
    {constructs}
  ]
  Links[
    {KnowledgeDomain bases Languagedesign},
    {Languagedesign isexpressed Grammars},
    {Grammars produces min 1 max 1 ProgramLanguage},
    {ComputerProgram isWritten ProgramLanguage},
    {ComputerProgram requires LanguageProc},
    {ProgramLanguage specifiedBy min 1 Grammars},
    {ProgramLanguage specifiedBy min 2 RegularExp},
    {CompilerConstTool constructs LanguageProc},
    {CompilerConstTool uses Grammars},
    {LexicalAnal uses RegularExp},
    {SyntacticAnal uses CFG},
    {SemanticAnal uses TG},
    {SemanticAnal uses min 1 max 2 AG},
    {CodeGen uses TG},
```

```
    {CodeGen uses AG},
    {TG implements max 2 SDT},
    {AG implements SemDT},
    {LanguageProc requires Analysis},
    {LanguageProc requires CodeGen}
  ]
}
```

Listing C.1: Language Processing OntoDL

C.2 LANGUAGE PROCESSING GENERATED OWL

Listing C.2 shows the OWL that is the outcome of Onto2OWL module.

```
<?xml version="1.0"?>
<Ontology xmlns="http://www.w3.org/2002/07/owl#"
     xml:base="http://example.com/onto2owl/Langprocessor/"
     {...}
    <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
    <Declaration>
        <Class IRI="#AG"/>
    </Declaration>
    <Declaration>
        <Class IRI="#Analysis"/>
    </Declaration>
    <Declaration>
        <Class IRI="#Analyzer"/>
    </Declaration>
    <Declaration>
        <Class IRI="#CFG"/>
    </Declaration>
    <Declaration>
        <Class IRI="#CodeGen"/>
    </Declaration>
    {...}
    <Declaration>
        <ObjectProperty IRI="#requires"/>
    </Declaration>
    <Declaration>
        <ObjectProperty IRI="#specifiedBy"/>
    </Declaration>
    <Declaration>
        <ObjectProperty IRI="#uses"/>
    </Declaration>
    <SubClassOf>
```

```
        <Class IRI="#Analysis"/>
        <Class IRI="#LexicalAnal"/>
    </SubClassOf>
    <SubClassOf>
        <Class IRI="#Analysis"/>
        <Class IRI="#SemanticAnal"/>
    </SubClassOf>
    <SubClassOf>
        <Class IRI="#Analysis"/>
        <Class IRI="#SyntacticAnal"/>
    </SubClassOf>
    <SubClassOf>
        <Class IRI="#Grammars"/>
        <Class IRI="#AG"/>
    </SubClassOf>
    {...}
    <SubClassOf>
        <Class IRI="#ProgramLanguage"/>
        <Class IRI="#DSL"/>
    </SubClassOf>
    <SubClassOf>
        <Class IRI="#ProgramLanguage"/>
        <Class IRI="#GPL"/>
    </SubClassOf>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#bases"/>
        <Class IRI="#KnowledgeDomain"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#isWritten"/>
        <Class IRI="#ComputerProgram"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#is_expressed"/>
        <Class IRI="#Languagedesign"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#produces"/>
        <Class IRI="#Grammars"/>
    </ObjectPropertyDomain>
    {...}
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#requires"/>
        <Class IRI="#ComputerProgram"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#requires"/>
```

```
        <Class IRI="#LanguageProc"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#specifiedBy"/>
        <Class IRI="#ProgramLanguage"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#uses"/>
        <Class IRI="#SyntacticAnal"/>
    </ObjectPropertyDomain>
    <ObjectPropertyRange>
        <Annotation>
            <AnnotationProperty abbreviatedIRI="owl:backwardCompatibleWith"/>
            <IRI>#KnowledgeDomain</IRI>
        </Annotation>
        <ObjectProperty IRI="#bases"/>
        <Class IRI="#Languagedesign"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <Annotation>
            <AnnotationProperty abbreviatedIRI="owl:backwardCompatibleWith"/>
            <IRI>#CompilerConstTool</IRI>
        </Annotation>
        <ObjectProperty IRI="#constructs"/>
        <Class IRI="#LanguageProc"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <Annotation>
            <AnnotationProperty abbreviatedIRI="owl:backwardCompatibleWith"/>
            <IRI>#AG</IRI>
        </Annotation>
        <ObjectProperty IRI="#implements"/>
        <Class IRI="#SemDT"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <Annotation>
            <AnnotationProperty abbreviatedIRI="owl:backwardCompatibleWith"/>
            <IRI>#TG</IRI>
        </Annotation>
        <ObjectProperty IRI="#implements"/>
        <ObjectMaxCardinality cardinality="2">
            <ObjectProperty IRI="#implements"/>
            <Class IRI="#SDT"/>
        </ObjectMaxCardinality>
    </ObjectPropertyRange>
    {...}
    <ObjectPropertyRange>
```

```
        <Annotation>
            <AnnotationProperty abbreviatedIRI="owl:backwardCompatibleWith"/>
            <IRI>#Grammars</IRI>
        </Annotation>
        <ObjectProperty IRI="#produces"/>
        <ObjectMinCardinality cardinality="1">
            <ObjectProperty IRI="#produces"/>
            <Class IRI="#ProgramLanguage"/>
        </ObjectMinCardinality>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <Annotation>
            <AnnotationProperty abbreviatedIRI="owl:backwardCompatibleWith"/>
            <IRI>#ProgramLanguage</IRI>
        </Annotation>
        <ObjectProperty IRI="#specifiedBy"/>
        <ObjectMinCardinality cardinality="1">
            <ObjectProperty IRI="#specifiedBy"/>
            <Class IRI="#Grammars"/>
        </ObjectMinCardinality>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <Annotation>
            <AnnotationProperty abbreviatedIRI="owl:backwardCompatibleWith"/>
            <IRI>#ProgramLanguage</IRI>
        </Annotation>
        <ObjectProperty IRI="#specifiedBy"/>
        <ObjectMinCardinality cardinality="2">
            <ObjectProperty IRI="#specifiedBy"/>
            <Class IRI="#RegularExp"/>
        </ObjectMinCardinality>
    </ObjectPropertyRange>
</Ontology>
```

Listing C.2: Language Processing OWL

C.3 LANGUAGE PROCESSOR GENERATED GRAMMAR

Listing C.3 show if the ontology is complex and extensive the resulting grammar will be more complex to understand.

C.3. Language Processor generated Grammar

```
grammar Langprocessor;
thing:
  'Thing['(codegen|regularexp|compilerconsttool|gpl|translator|ag|analyzer|cfg|
    semanticanal|interpreter|sdt|languagedesign|compilers|knowledgedomain|
    computerprogram|dsl|syntacticanal|tg|semdt|lexicalanal)+ ']'
;


codegen:  'CodeGen{'codegenID
  (',' '[' (codegen_uses_ag) (codegen_uses_ag)?(codegen_uses_tg)* ']' )?'}'
;

codegenID :
   STRING
;
codegen_uses_ag:
  '{' 'uses' 'ag' agID'}';
codegen_uses_tg:
  '{' 'uses' 'tg' tgID'}';

regularexp: 'RegularExp{'regularexpID '}'
;

regularexpID :
   STRING
;

compilerconsttool:  'CompilerConstTool{'compilerconsttoolID
  (',' '[' (compilerconsttool_uses_grammars)*(
    compilerconsttool_constructs_languageproc)* ']' )?'}'
;

compilerconsttoolID :
   STRING
;
compilerconsttool_uses_grammars:
  '{' 'uses' 'grammars' grammarsID'}';
compilerconsttool_constructs_languageproc:
  '{' 'constructs' 'languageproc' languageprocID'}';

gpl:  'GPL{'gplID
  (',' '[' (programlanguage)+
 ']' )?'}'
;

gplID :
   STRING
```

```
;

programlanguage:  'ProgramLanguage{'programlanguageID
  (',' '[' (programlanguage_specifiedby_grammars)+(
    programlanguage_specifiedby_regularexp) (
    programlanguage_specifiedby_regularexp)+ ']' )?'}'
;

programlanguageID :
   STRING
;
programlanguage_specifiedby_grammars:
  '{' 'specifiedby' 'grammars' grammarsID'}';
programlanguage_specifiedby_regularexp:
  '{' 'specifiedby' 'regularexp' regularexpID'}';

translator: 'Translator{'translatorID
  (',' '[' (languageproc)+
 ']' )?'}'
;

translatorID :
   STRING
;

languageproc: 'LanguageProc{'languageprocID
  (',' '[' (languageproc_requires_analysis)*(languageproc_requires_codegen)* ']
    ' )?'}'
;

languageprocID :
   STRING
;
languageproc_requires_analysis:
  '{' 'requires' 'analysis' analysisID'}';
languageproc_requires_codegen:
  '{' 'requires' 'codegen' codegenID'}';

ag: 'AG{'agID
  (',' '[' (grammars)+
 ']' )?
  (',' '[' (ag_implements_semdt)* ']' )?'}'
;

agID :
   STRING
;
```

```
ag_implements_semdt:
  '{' 'implements' 'semdt' semdtID'}';

grammars: 'Grammars{'grammarsID
  (',' '[' (grammars_produces_programlanguage) ']' )?'}'
;

grammarsID :
   STRING
;
grammars_produces_programlanguage:
  '{' 'produces' 'programlanguage' programlanguageID'}';

analyzer: 'Analyzer{'analyzerID '}'
;

analyzerID :
   STRING
;

cfg:  'CFG{'cfgID '}'
;

cfgID :
   STRING
;

semanticanal: 'SemanticAnal{'semanticanalID
  (',' '[' (analysis)+
 ']' )?
  (',' '[' (semanticanal_uses_tg)*(semanticanal_uses_ag) (semanticanal_uses_ag)
    ? ']' )?'}'
;

semanticanalID :
   STRING
;
semanticanal_uses_tg:
  '{' 'uses' 'tg' tgID'}';
semanticanal_uses_ag:
  '{' 'uses' 'ag' agID'}';

analysis: 'Analysis{'analysisID '}'
;

analysisID :
   STRING
```

```
;

interpreter:  'Interpreter{'interpreterID '}'
;

interpreterID :
   STRING
;

sdt:  'SDT{'sdtID '}'
;

sdtID :
   STRING
;

languagedesign: 'Languagedesign{'languagedesignID
  (',' '[' (languagedesign_is_expressed_grammars)* ']' )?'}'
;

languagedesignID :
   STRING
;
languagedesign_is_expressed_grammars:
  '{' 'is_expressed' 'grammars' grammarsID'}';

compilers:  'Compilers{'compilersID '}'
;

compilersID :
   STRING
;

knowledgedomain:  'KnowledgeDomain{'knowledgedomainID
  (',' '[' (knowledgedomain_bases_languagedesign)* ']' )?'}'
;

knowledgedomainID :
   STRING
;
knowledgedomain_bases_languagedesign:
  '{' 'bases' 'languagedesign' languagedesignID'}';

computerprogram:  'ComputerProgram{'computerprogramID
  (',' '[' (computerprogram_requires_languageproc)*(
    computerprogram_iswritten_programlanguage)* ']' )?'}'
;
```

```
computerprogramID :
   STRING
;
computerprogram_requires_languageproc:
  '{' 'requires' 'languageproc' languageprocID'}';
computerprogram_iswritten_programlanguage:
  '{' 'iswritten' 'programlanguage' programlanguageID'}';

dsl:  'DSL{'dslID '}'
;

dslID :
   STRING
;

syntacticanal:  'SyntacticAnal{'syntacticanalID
  (',' '[' (syntacticanal_uses_cfg)* ']' )?'}'
;

syntacticanalID :
   STRING
;
syntacticanal_uses_cfg:
  '{' 'uses' 'cfg' cfgID'}';

tg: 'TG{'tgID
  (',' '[' (tg_implements_sdt)? (tg_implements_sdt)? ']' )?'}'
;

tgID :
   STRING
;
tg_implements_sdt:
  '{' 'implements' 'sdt' sdtID'}';

semdt:  'SemDT{'semdtID '}'
;

semdtID :
   STRING
;

lexicalanal:  'LexicalAnal{'lexicalanalID
  (',' '[' (lexicalanal_uses_regularexp)* ']' )?'}'
;
```

```
lexicalanalID :
   STRING
;
lexicalanal_uses_regularexp:
  '{' 'uses' 'regularexp' regularexpID'}';
```

Listing C.3: Language Processor outcome from OWL2DSL

C.4 LANGUAGE PROCESSING DDESC INPUT

In this next Listing it will be presented a possible input for the DDesc2OWL Module and it will show example of hierarchical relations.

```
Thing[
  TG{
    "Intance of tg ID"
    ,[
      Grammars{  "Intance of grammars ID"
        ,[  { produces programlanguage "programlanguageID_reference" } ]
      }
    ]
    ,[
      { implements sdt "sdtID_reference1" }
      { implements sdt "sdtID_reference2" }
    ]
  }
  SDT{   "Intance of sdt ID"
  }
  GPL{
    "Intance of gpl ID"
    ,[
      ProgramLanguage{
        "Intance of programlanguage ID"
        ,[
          { specifiedby grammars "grammarsID_reference" }
          { specifiedby regularexp "regularexpID_reference1" }
          { specifiedby regularexp "regularexpID_reference2" }
        ]
      }
    ]
  }
  RegularExp{  "Intance of regularexp ID"
  }
  CFG{   "Intance of cfg ID"
```

```
    }
]
```

Listing C.4: DDesc input file for Ddesc2OWL module

C.5 LANGUAGE PROCESSING DDESC2OWL RESULT

Figure 22 represent the outcome of the Ddesc2OWL Module.

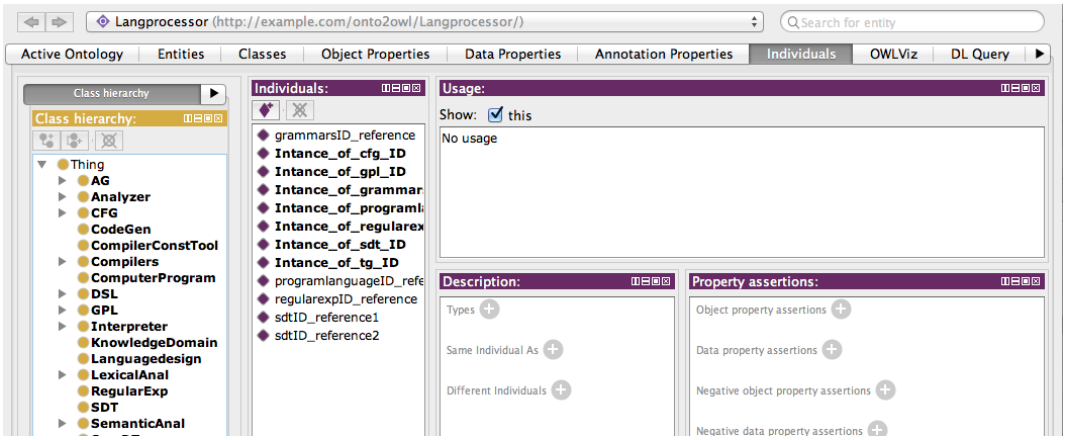

Figure 22.: DDesc2OWL outcome opened in Protégé