ALMA MATER STUDIORUM-UNIVERSITY OF BOLOGNA

FACULTY OF ENGINEERING

2nd LEVEL DEGREE COURSE IN
COMPUTER SCIENCE ENGINEERING

THESIS IN ARTIFICIAL INTELLIGENCE

ONTOCREATOR: DESIGN AND IMPLEMENTATION OF A TOOL FOR THE AUTOMATIC CREATION OF OWL1.1 ONTOLOGY

CANDIDATE:

Cappa Sebastiano

COMMITTEE OF ULM UNIVERSITY:

Prof. Dr. Friedrich von Henke
Timo Weithöner

COMMITTEE OF BOLOGNA UNIVERSITY:

Prof. Paola Mello
Eng. Federico Chesani

Academic Year [2007/08]

Session II

18 July 2008

to all the people
who have accomplish me
to realize this thesis

Owl
Ontology
Java Web Start
Reasoner-Benchmark
Semantic application

Contents

1 Problems and motivation

Today there are more than one billion (1,131 billion in 2006) users [1] that use 168 million web servers [2] of the World Wide Web. Therefore, there are some serious problems such as finding, extracting, representing and interpreting the information. These problems are due to the fact that information processing is almost purely syntactic. This means that there is no understanding of the meaning of words and sentences, that is the *semantic*.

The future of WWW is evolving from syntactic-mode to semantic-mode using the meaning expressed by syntax. The solution proposed by the World Wide Web Consortium (W3C[1]) for the next generation of the web is the Semantic Web [3]. A key element of this technology is that applications can determine the meaning of text and create connections between the entities. In semantic applications the information is well defined and depends on the context; so, people and computers can work in cooperation toward the same objective. To combine information from different applications, the data must be compatible and intelligible by machines and applications. Consequently, it is necessary to have a standard language to represent semantic information of *ontologies*[2], like *OWL*.

OWL (Web Ontology Language [4]) is designed to exchange ontologies and to be used by applications that need to process the content of information. With OWL, applications can apply automatic reasoning over ontologies. Applications providing these reasoning services are called *ontology reasoners* or *reasoners*.

Benchmarks[3] check the soundness, completeness and performance of reasoners. Obviously, they need to have many ontologies to verify.

Unfortunately, there is now a low quantity of ontologies available, especially for huge ontologies with interconnected data and with all types of possible entities. The problem is that writing a huge ontology is complex and time-consuming. So, it is necessary to have a tool that produces a solution to this problem.

Currently there are some tools able to create huge ontologies, e.g.

[1] http://www.w3.org/

[2] In computer science ontologies are representations of a set of concepts within a domain and relationships between individuals.

[3] In computing, a benchmark is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of a system.

the Lehigh University Benchmark (LUBM[4]), but they have many limits. For example, they have problems with sparsely interrelated data, with memory allocation and with OWL DL full coverage. This argument will be discussed in more details in Section 2.4.

These shortcomings motivated us to produce a new tool that resolves this problem.

This document is further structured as follows. In Chapter 2 the thesis gives a background about theories on semantic area and especially on OWL.

Then, it explains the details of OntoCreator in Chapter 3. In particular, the thesis discusses the goal and the requirements of the tool. Next, it introduces the problems encountered in the design phase. After this, the thesis debates the choices regarding those problems. Then, it explains the architecture of the tool and the implementation.

In Chapter 4 there is the evaluation of the tool through tests and comparisons between the different possible implementations. The goal of this section is to improve the computational performance. In the last section of this chapter, the thesis discusses the consistency's evaluation of the ontologies created by the tool.

Finally, in Chapter 5, the document presents conclusions and possible future works.

[4]http://swat.cse.lehigh.edu/projects/lubm/

2 Introduction

This chapter presents an introduction on ontologies and OWL. It describes the features of ontologies and different types of OWL. It also deals with reasoners and benchmarks, explaining the distinctive characteristics of them. The goal of this chapter is to give a background on the subject of this thesis.

2.1 OWL

In computer science, ontologies are representations of a set of *Concepts* within a domain and *Properties* between *Individuals*. Ontologies can be written in many different syntactic forms. As already mentioned, OWL is the W3C standard for representing ontologies. The OWL language is a semantic markup language for publishing and sharing ontologies. It was developed as an extension of RDF (the Resource Description Framework[5]).

RDF is a family of specifications to model a variety of syntax formats. The RDF model is based on the idea of making statements about resources in the form of subject-predicate-object expressions, called triples in RDF terminology. RDFS or RDF Schema is an extensible knowledge representation language, providing basic elements for the description of ontologies.

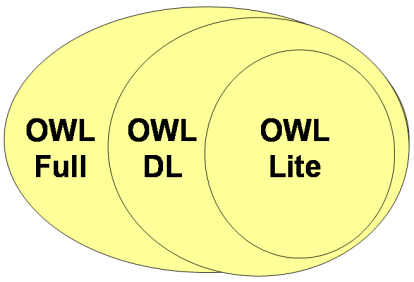

Figure 1: OWL, sub-languages hierarchies

[5]http://www.w3.org/RDF/

In Figure 1, it can be seen that OWL provides three increasingly-expressive sub-languages: *OWL Lite, OWL DL, and OWL Full.* They offer increasing levels of expressivity.

The Full version is the most complete (with maximum expressiveness) sub-language, but it has no computational guarantees.

OWL-DL, a variant that is based on a family of description logics (DL), facilitates the description of complex concepts. In the OWL DL, a class cannot be an instance of another class. It has computational completeness (all conclusions are guaranteed to be computable) and decisiveness (all computations will finish in finite time). In other words, a feature of OWL-DL is that reasoning tasks terminate after a finite amount of time and that the inferences drawn are valid.

OWL Lite has a lower formal complexity than OWL DL. For example, the cardinality constraints can only have the values zero or one.

Each of these sub-languages is an extension of its predecessor. This means that an OWL Lite ontology is also an OWL DL ontology (idem for Lite to Full and DL to Full).

There are two versions of OWL DL; the first is OWL1.0[5] which was released on 29 July 2002.

From December 19th 2006 the W3C has been working on new OWL1.1[6] specification to produce a W3C Recommendation.

OWL1.1, like OWL1.0, is compatible with OWL-DL and OWL-Full. The important features of OWL 1.1 are that it has new DL constructs (like qualified cardinality restrictions or reflexive, irreflexive, anti-symmetric and disjoint properties).

On 11th April 2008, the W3C released a new document in which it was explained that the name of the language had been changed from OWL 1.1 to OWL 2[7] since the previous Working Draft (dated 8 January 2008).

The requirement for this thesis was to use OWL DL, in particular OWL1.1.

2.2 Ontology

This section describes the features of ontologies in OWL.

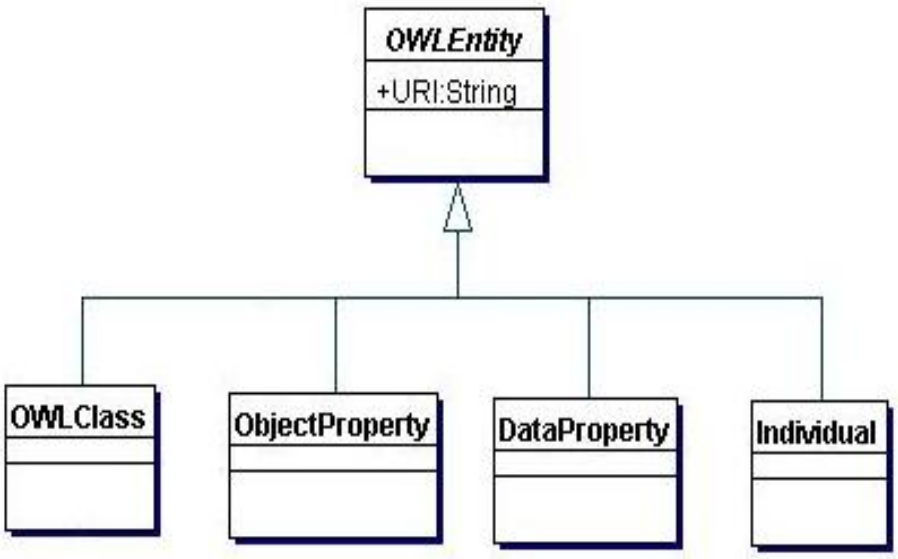

Figure 2: OWL, entity hierarchy

Concepts, Properties and *Individuals* are the fundamental building blocks of ontologies. Their hierarchy is described in Figure 2[6]. Here it can be seen that Classes (that are Concepts), Properties and Individuals are *OWLEnties*. An Entity has a URI that represents the name.

A Class is a group of Individuals. Usually, the Individuals are in the same Class because they have common features.

Classes can have Sub-Classes that represent a subset of Individuals. Classes can also be related to other Classes by *Equivalent* or *Disjoint Axioms*.

[6]like all figures in this section, its origin is: *OWL 1.1 Web Ontology Language Structural Specification and Functional-Style Syntax.* Copyright 2006-2007 by the Authors [Boris Motik, The University of Manchester; Peter F. Patel-Schneider, Bell Labs Research and Lucent Technologies; Ian Horrocks, The University of Manchester]. http://www.webont.org/owl/1.1/owl_specification.html

Figure 3 shows an example of a classes' hierarchy.

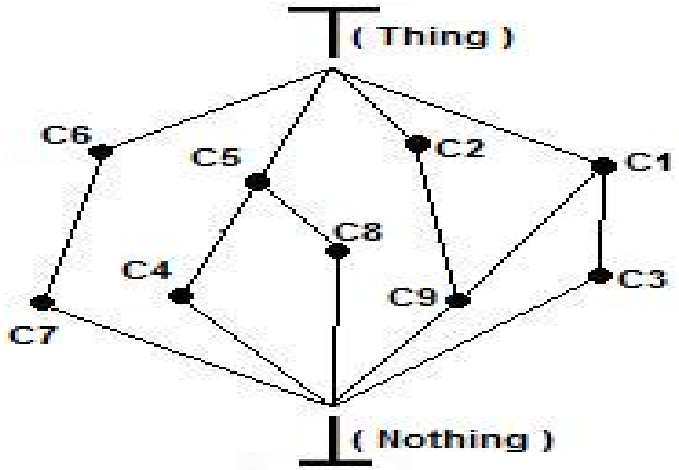

Figure 3: OWL, classes hierarchy

All the Classes are a subclass of *Thing*($\top$) that represents the general Concept. Then, a Class defines a group of Individuals of *Thing*. In the other side, the class *Nothing*($\bot$) represents the empty Class which is the common subclass of all the Classes.

ObjectProperty is a Property which expresses a relation between Individuals. The *Domain* of an ObjectProperty is the Class where the Property is applied. The *Range* is the Class that is the destination of the Property.

DataProperty is a Property connected with a simple data. The Domain of a DataProperty is again a Class. Instead, differently from ObjectProperty, the Rangeof a DataProperty is a xml datatype. We are not concerned with in this thesis to DataProperty.

An Individual is a member of one or more Classes and it can be described as equal to or different from other Individuals.

An ontology can also have *Constants*, *Annotations* and *Datatypes*, but they are not dealt with this thesis.

2.2.1 Concepts

As explained in the previous section, ontologies are characterized by Concepts that are expressed by Classes. Ontologies can have many different types of Classes. For example, Classes can be further defined by specifications such as *Union, Intersection, One-Of* or *Complement* of other Classes.

The Classes connections are explained in Figure 4.

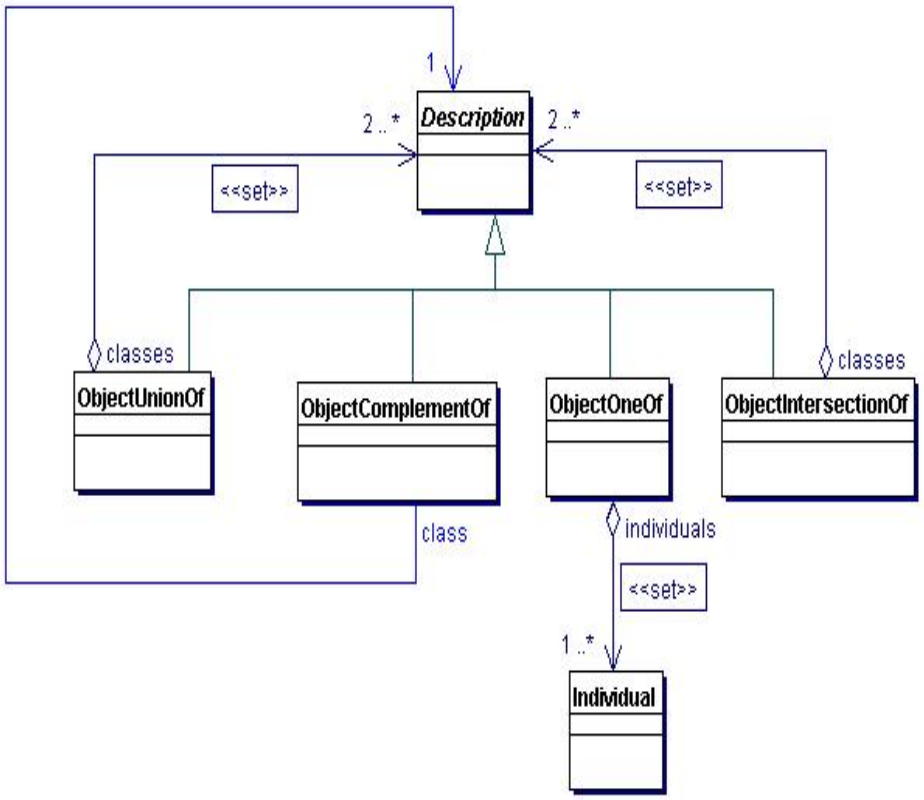

Figure 4: OWL, classes connections

In Figure 4 there are:

- *UnionOf(C_1, C_2... C_n)*: is the aggregation of Individuals belonging to the set of Classes that are united in the Concept. The number of Classes united is a minimum of two. UnionOf is a *Disjunction* boolean combination.
- *ComplementOf(C)*: are all Individuals of the ontology which do not belong to a given Class *C*.
- *OneOf(a_1...a_n)*: is a list of one or more Individuals (*Nominal* combination).
- *IntersectionOf(C_1, C_2 ... C_n)*: are all Individuals which belong simultaneously to all given Classes. Again the number of Classes is a minimum of two. IntersectionOf is a *Conjunction* boolean combination.

In Table 1 there is a pseudo-DL[7] interpretation of Classes.

Table 1: Classes interpretation

ObjectComplementOf(C)	$\top \setminus C$
ObjectIntersectionOf(C_1, C_2 ... C_n)	$C_1 \cap C_2 \cap ... \cap C_n$
ObjectUnionOf(C_1, C_2... C_n)	$C_1 \cup C_2 \cup ... \cup C_n$
ObjectOneOf(a_1...a_n)	$a_1, ..., a_n$

[7] *C* indicates a Class, *R* indicates a Property, *a* indicates an Individual, $\top$ is the Class *Thing*, ♯ means the number of instances that satisfy what it is inside the braces.

Then, there are Classes defined by restrictions on object Properties. They are explained in Figure 5.

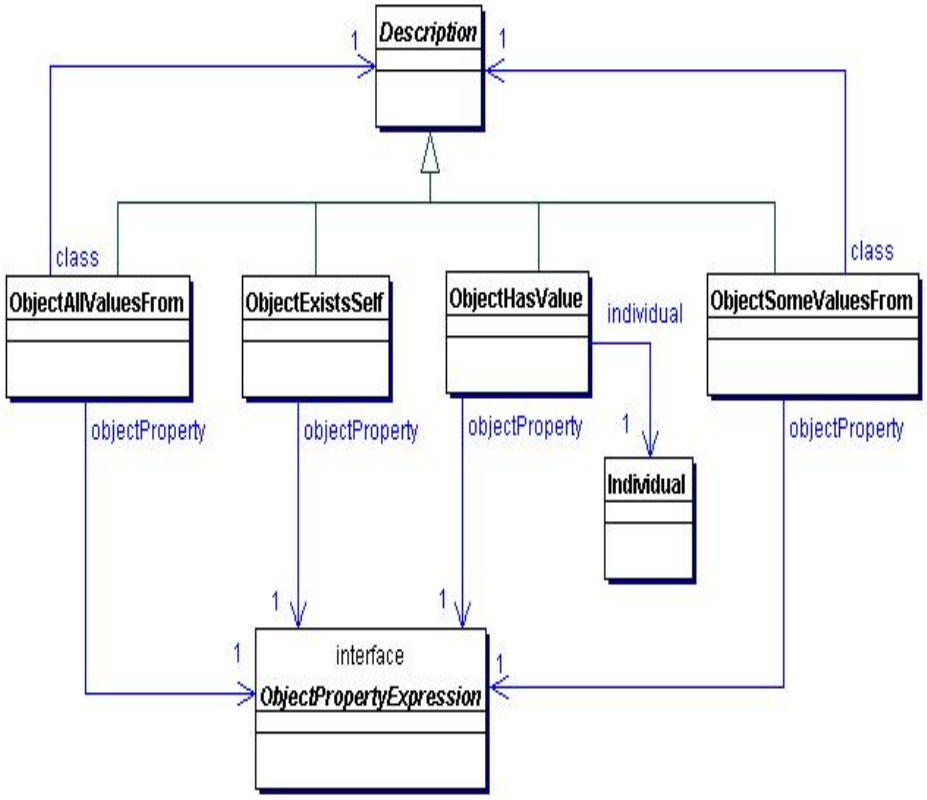

Figure 5: OWL, Classes defined by Properties

As we can see here, unlike those previously mentioned, these Classes use a Property to describe the Individuals who belong to them.

These Classes are:

- *AllValuesFrom(R, C)*: indicates all Individuals that have references, by a given Property R, only to instances of a given Class C.

- *ExistsSelf(R)*: indicates all Individuals connected to themselves by a given Property R.

- *HasValue(R, a)*: Defines a Class consisting of all Individuals which are connected to a given Individual a through a given Property R.

- *SomeValuesFrom(R, C)*: indicates all Individuals having a minimum of one reference to an instance of a given Class C by a given Property R.

In Table 2 there is a pseudo-DL interpretation of those Classes.

Table 2: Classes interpretation

ObjectSomeValuesFrom(R, C)	$x \mid \exists\, y : (x, y) \in R$ and $y \in C$
ObjectAllValuesFrom(R, C)	$x \mid \forall\, y : (x, y) \in R \rightarrow y \in C$
ObjectHasValue(R, a)	$x \mid (x, a) \in R$
ObjectExistsSelf(R)	$x \mid (x, x) \in R$

Sub-Class, Equivalent-Class and *Disjoint-Class* are Axioms. The difference between entity description and Axiom is that a description defines an entity (Class or Property or Individual), while an Axiom identifies a statement about an entity.

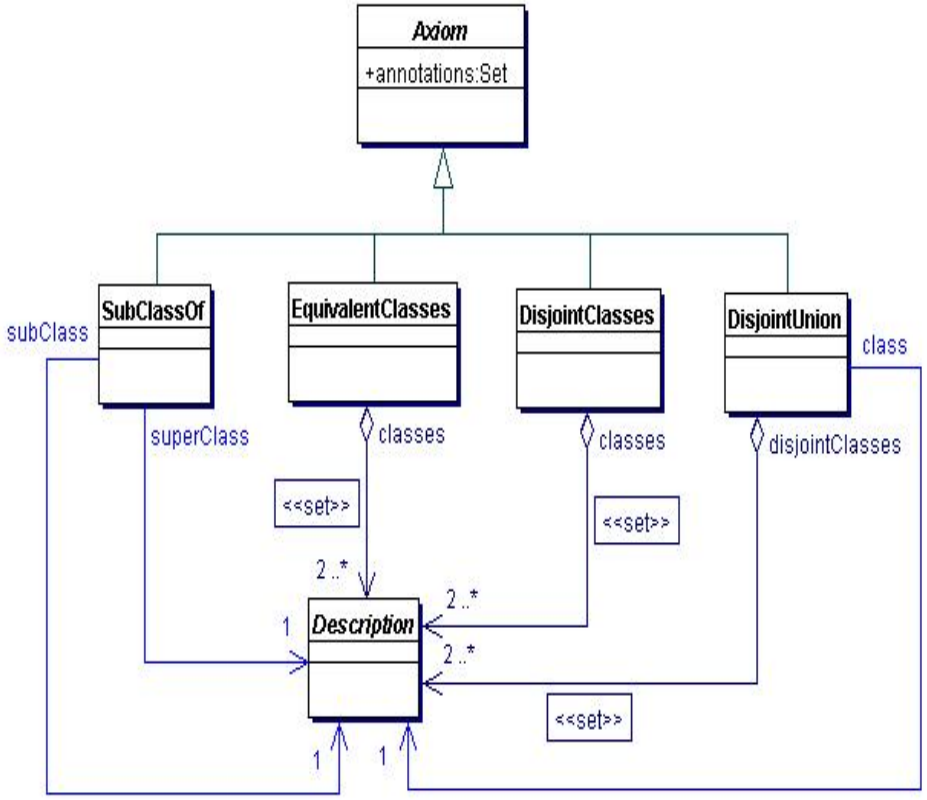

Figure 6: OWL, Classes Axioms

Disjoint-Union Classes are not explicitly covered in this project because they can be viewed as the combination of Disjoint Classes and Union Classes.

In Figure 6 there are:

1. *Equivalent Classes*, when two or more Classes describe the same concept;
2. *Disjoint Classes*, when two or more Classes have no Individuals in common;
3. *Sub Classes*, when a class is a subset of the Individuals who belong to another Class.

In Table 3 there is a pseudo-DL interpretation of Class Axioms.

Table 3: Class Axioms interpretation

SubClassOf(C,D)	$C \subseteq D$
EquivalentClasses(C_1 ... C_n)	$C_1 = ... = C_n$
DisjointClasses(C_1 ... C_n)	$C_1 \cap ... \cap C_n = 0$

Another important restriction is the *Cardinality*.

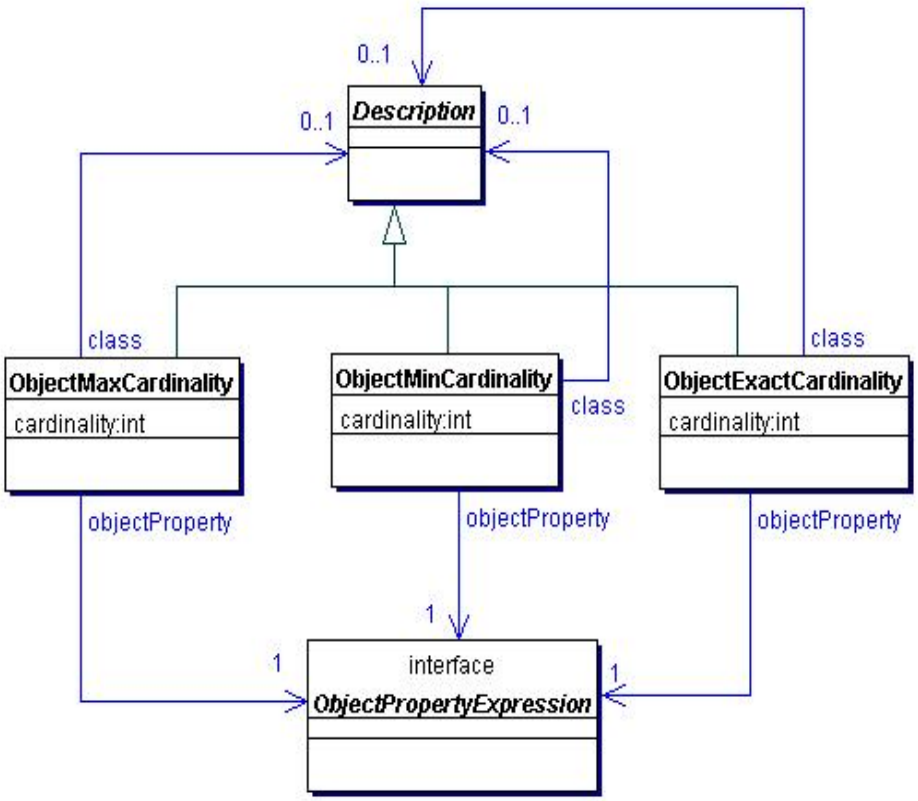

Figure 7: OWL, cardinality restriction

Figure 7 shows the three different types of cardinality restriction: *Min, Max* and *Exact*.

Min Cardinality Concepts assume that all the Individuals belonging to them have a minimum number of relations through the given Property with the given Class. By contrast, *Max Cardinality* requires that the number of relations are less than a maximum number.

Exact Cardinality restriction is the combination of *Min Cardinality* restriction and *Max Cardinality* restriction. Indeed, the number of relations must be equal to the value of Cardinality.

In Table 4 there is a pseudo-DL interpretation of Classes defined by *Qualified Cardinality* restriction.

Table 4: Qualified Cardinality restriction interpretation

ObjectMinCardinality(n, R, C)	$x \mid \sharp\, y \mid (x, y) \in R \text{ and } y \in C \leq n$
ObjectMaxCardinality(n, R, C)	$x \mid \sharp\, y \mid (x, y) \in R \text{ and } y \in C \geq n$
ObjectExactCardinality(n, R, C)	$x \mid \sharp\, y \mid (x, y) \in R \text{ and } y \in C = n$

If the cardinality restriction is not defined in a Class, the range of the Property is all the ontology, i.e. the class *Thing* ($\top$). In Table 5 there is a pseudo-DL interpretation of Classes defined by *Unqualified Cardinality* restriction.

Table 5: Unqualified Cardinality restriction interpretation

ObjectMinCardinality(n, R)	$x \mid \sharp\, y \mid (x, y) \in R \text{ and } y \in \top \leq n$
ObjectMaxCardinality(n, R)	$x \mid \sharp\, y \mid (x, y) \in R \text{ and } y \in \top \geq n$
ObjectExactCardinality(n, R)	$x \mid \sharp\, y \mid (x, y) \in R \text{ and } y \in \top = n$

2.2.2 Properties

Another important element in ontologies is the Property. It expresses relationships between Individuals. A Property has a Domain that indicates the Classes to which it can be applied. A Property also has a Range that is the destination of the Property.

An important difference between Classes and Properties is that all Classes have similar ancestor (the top Class *Thing*) and similar Sub Class (the bottom Class No*Thing*), while the Properties do not have a constraint legacy.

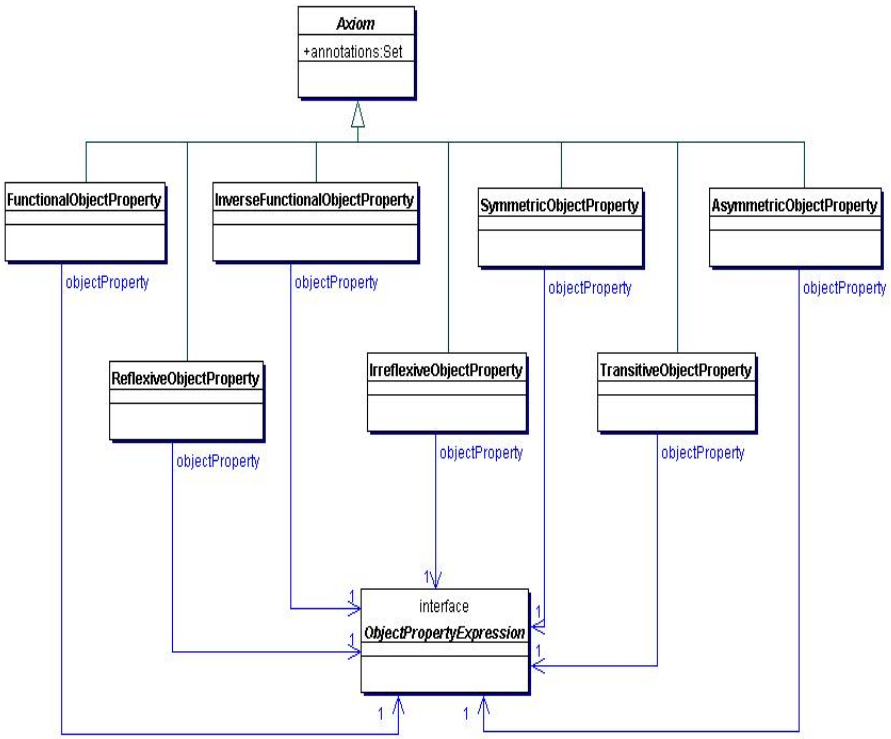

Figure 8: OWL, Properties Axioms

In Figure 8, there are *Functional, InverseFunctional, Symmetric, Asymmetric, Reflexive, Irreflexive* and *Transitive* Properties. As we can see, these entities are Axioms and they have a connection with one Property.

A short description of them follows:

1. *Inverse*: if P1 is the inverse of P2, all pairs of Individuals that are connected by P2, are also connected by P1 with Domain and Range reverse;

2. *Functional*: if it is applied to two pairs of Individuals and these two pairs have the same Individual in the Domain, this means that the two Individuals of that Range are the same Individual. So, a Property connects Individuals of Domain with no more than one Individual of Range (the Max Cardinality is one);

3. *Inverse-functional*: if it is applied to two pairs of Individuals with the same Individual of the Range, this means that the two Individuals of the Domain are the same Individual;

4. *Reflexive*: this means that the Individuals are connected with themselves;

5. *Irreflexive*: this means that the Individuals are not connected with themselves;

6. *Symmetric*: if applied to a pair of Individuals (x,y) this means that also the pair (y,x) has this Property;

7. *Antisymmetric*: if it is applied to a pair of Individuals (x,y) this means that the pair (y,x) does not have not this Property;

8. *Transitive*: if it is applied to two pairs of Individuals (x,y) and (y,z) this means that the pair (x,z) also has this Property;

There are also:

1. *Sub-Properties*: all the pairs of a Sub-Property are also connected by another Property that represents the super-set of the Sub-Property;
2. *Equivalent-Properties*: if a Property is equal to another Property;
3. *Disjoint-Properties*: if P1 is disjoint with P2, all pairs of Individuals connected by P1 are not connected by P2, and conversely;

The semantic of the mentioned elements is shown in Table 6.

Table 6: OWL, interpreting Properties

ObjectPropertyDomain(R, C)	$x \mid \exists\, y : (x, y) \in R \subseteq C$
ObjectPropertyRange(R, C)	$y \mid \exists\, y : (x, y) \in R \subseteq C$
InverseObjectProperties(R, S)	$R = (x, y) \mid (y, x) \in S$
FunctionalObjectProperty(R)	$(x, y_1) \in R$ and $(x, y_2) \in R \rightarrow y1 = y2$
InverseFunctionalObjectProperty(R)	$(x_1, y) \in R$ and $(x_2, y) \in R \rightarrow x1 = x2$
ReflexiveObjectProperty(R)	$x \in \top \rightarrow (x, x) \in R$
IrreflexiveObjectProperty(R)	$x \in \top \rightarrow (x, x)$ is not in R
SymmetricObjectProperty(R)	$(x, y) \in R \rightarrow (y, x) \in R$
AntisymmetricObjectProperty(R)	$(x, y) \in R \rightarrow (y, x)$ is not in R
TransitiveObjectProperty(R)	$(x, y) \in R$ and $(y, z) \in R \rightarrow (x, z) \in R$
SubObjectPropertyOf(R, S)	$R \subseteq S$
EquivalentObjectProperties(R_1 ... R_n)	$R_1 = ... = R_n$
DisjointObjectProperties(R_1 ... R_n)	$R_1 \cap ... \cap R_n = 0$

2.2.3 Individuals

The third important element in ontologies is Individual. Individuals are instances of Classes and they are related to other Individuals by Properties.

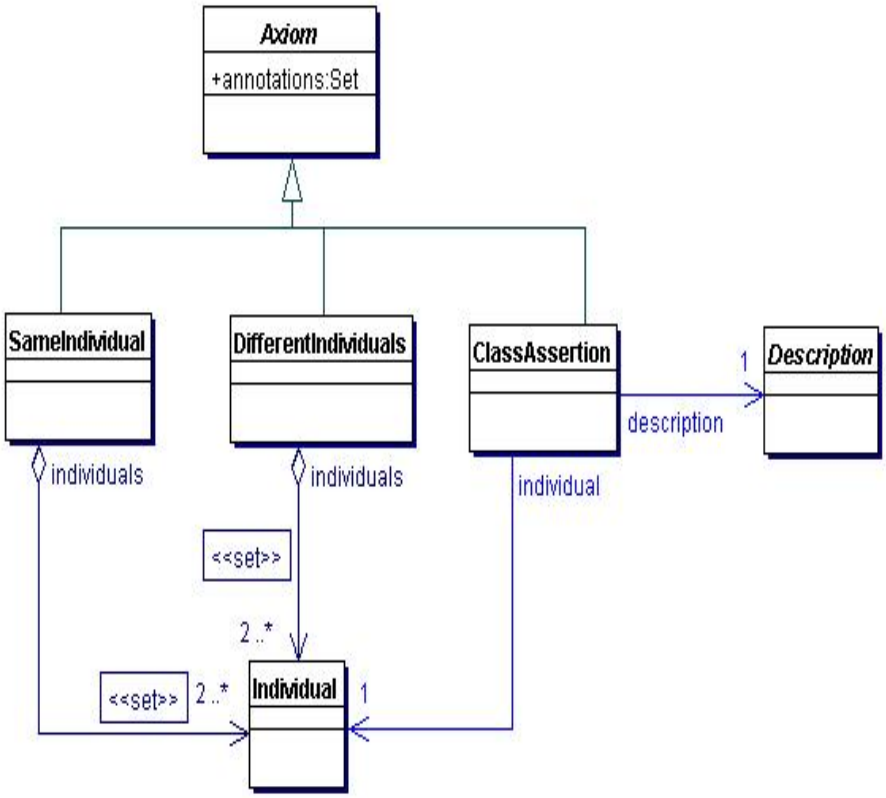

Figure 9: OWL, Individuals

An Individual can be the same (*SameIndividual*) as other Individuals and different (*DifferentIndividual*) from other Individuals. In Figure 9 we can see that SameIndividual and DifferentIndividual are Axioms and that they need at least two Individuals.

A *Class Assertion* declares that one Individual belongs to a Class.

2.2.4 Consistency

An ontology is *consistent* if it is not contradictory, or if we cannot demonstrate an Axiom and its negation. Inconsistent ontologies have a contradiction between their entities, such as an instance of an *unsatisfiable/incoherent* Class.

The consistency depends on the relationship between Concepts, Properties, Individuals. These dependencies must be respected and they should not introduce contradictions to ensure the consistency of the semantics. Therefore, the risk of inconsistency increases with the number of Axioms because they add relationships between the entities.

An ontology can be consistent but it can have unsatisfiable Classes. Unsatisfiable Classes are those that cannot have any possible individual. They must be equivalent to the empty set, otherwise the ontology is inconsistent.

2.3 DL reasoners

In this section, the thesis discusses the state of the art of Description Logic reasoners, namely: FaCT++, RACER, Pellet, OWLIM and KAON2. The goal of this section is to evaluate which reasoners we can use in this thesis.

2.3.1 FaCT++

FaCT++[8] is a reasoner for Description Logic languages. It was developed at the University of Manchester.

FaCT++ is a C++ re-implementation of the DL reasoner FaCT[9], which has been implemented in Lisp. This new version of FaCT is more effective and portable thanks to the programming language and a different internal architecture[8]. FaCT++ uses optimised tableaux algorithms.

The latest available version (1.1.11) was released on March 28, 2008 and it is distributed under GPL[10] license.

It supports OWL1.1, but the data types supported by FaCT++ are just strings and integers. Therefore, it has no full data types support.

2.3.2 OWLIM

OWLIM[11] was developed by OntoText Lab and the first version appeared in 2005. It is a high-performance semantic repository developed in Java and it is based on TRREE[12], a native RDF rule-entailment engine.

OWLIM is an efficient tool for querying the Sesame[13] RDF database. Indeed, developers benefit for high level of query languages bundled with the Sesame platform.

The latest available version is the 2.9.1, released on September 10, 2007. It is is an open-source Java library available under LGPL[14]

[8] http://owl.man.ac.uk/factplusplus/

[9] http://www.cs.man.ac.uk/ horrocks/FaCT/

[10] The General Public License is a free software license originally written by Richard Stallman for the GNU project.

[11] http://www.ontotext.com/owlim/index.html

[12] http://www.ontotext.com/trree/index.html

[13] http://www.openrdf.org/doc/sesame2/2.1.2/users/index.html

[14] The Lesser General Public License is a modified, more permissive, version of the GPL.

license.

The most important advantage of OWLIM is that it can manage huge ontologies with millions of entities.

A disadvantage of OWLIM is that it supports only OWL Lite, and not OWL DL.

2.3.3 Racer

Racer[15] (Renamed Abox and Concept Expression Reasoner) was developed by Prof. Dr. Volker Haarslev at Concordia University of Montreal and by Prof. Dr. Ralf Mllerat at Hamburg University of Technology. The project began in 1997 and the first version of the tool appeared in 2002. In fact, Racer was the first OWL Reasoner.

Racer is now called RacerPro and is being continuously improved. It is written in Common Lisp and it incorporates all optimization techniques of FaCT.

RacerPro supports all types of OWL, including OWL1.1 and it can handle datatypes, namely natural, integer, real and complex numbers, and strings. It can also be used by TCP communication.

RacerPro is only provided free of charge for universities and research labs, and the source code is not publicly available.

The latest version available is the 1.9.2, released on October 24, 2007.

2.3.4 Pellet

Pellet[16] was developed at the University of Maryland. It is implemented in Java, and the source code is freely available. It has an open source version and one commercially supported by Clark-Parsia LLC[17].

Pellet supports OWL DL and from the version 1.4 it includes OWL1.1. It is based on the tableaux algorithms.

On 1st May 2008 a new version of Pellet (1.5.2) was released under MIT license[18].

[15] http://www.racer-systems.com/

[16] http://pellet.owldl.com/

[17] http://clarkparsia.com/

[18] The MIT license is a particular free software license. It permits the reuse and the redistribution of a software without warranty of any kind.

The main advantage of Pellet is its support of xml Schema data types.

2.3.5 KAON2

KAON2[19], which it means *Karlsruhe ontology*, was developed at the University of Karlsruhe by Boris Motik and it has now based at the University of Manchester. It is a successor to the KAON project. KAON used a proprietary extension of RDFS, whereas KAON2 is based on OWL-DL and F-Logic. It is a completely new system, and is not backward-compatible with KAON.

In 2005, the first version of KAON2 was released; while the latest version was released on January 14, 2008. The tool is available under Pay Licensed Closed Source[20].

Contrary to most currently available DL reasoners, such as FaCT, FaCT++, Racer or Pellet, KAON2 does not implement the tableaux calculus. Rather, reasoning in KAON2 is implemented by novel algorithms which reduce a SHIQ(D) knowledge base to a disjunctive datalog program.

The disadvantages are that KAON2 cannot currently handle nominals. So, if an ontology contains an owl:oneOf class or an owl:hasValue restriction, each reasoning task will throw an error. In addition, KAON2 cannot currently handle large numbers in cardinality statements.

2.3.6 Summary

In this chapter, we presented the most important OWL reasoners. This introduction was necessary to understand which reasoners we can use and which problems we are going to encounter using them.

In the thesis, we are going to use Racer and Pellet to check the ontology produced by the tool. The reasons for this choice are that they support OWL1.1, are open source and have a good documentation.

[19] http://kaon2.semanticweb.org/

[20] Pay Licensed Closed Source is a category of commercial software licenses. It presupposes to pay to use the software and it prohibits to view the source code.

2.4 Benchmarks

In the previous section, the document discussed reasoners, which special reference to their problems. In this section, the thesis presents the benchmarks and their limits.

The scope of benchmarks is to check applications and then to provide helpful hints for developers to improve the software. Benchmarks check the soundness, completeness and performance of reasoners.

One of the most popular benchmarks for OWL Knowledge Base Systems is LUBM[21] (Lehigh University Benchmark). LUBM is intended to evaluate the performance of huge OWL repositories. It can also create realistic ontology over university domain.

The limits of LUBM[9] are that its ontologies have sparsely interrelated data and inference on cardinality and allValueFrom restrictions cannot be tested by the LUBM. In fact, the inference supported by this benchmark is only a subset of OWL Lite.

A direct extension of the LUBM in terms of expressiveness is University Ontology Benchmark (UOBM). It includes a covering of OWL Lite and OWL DL.

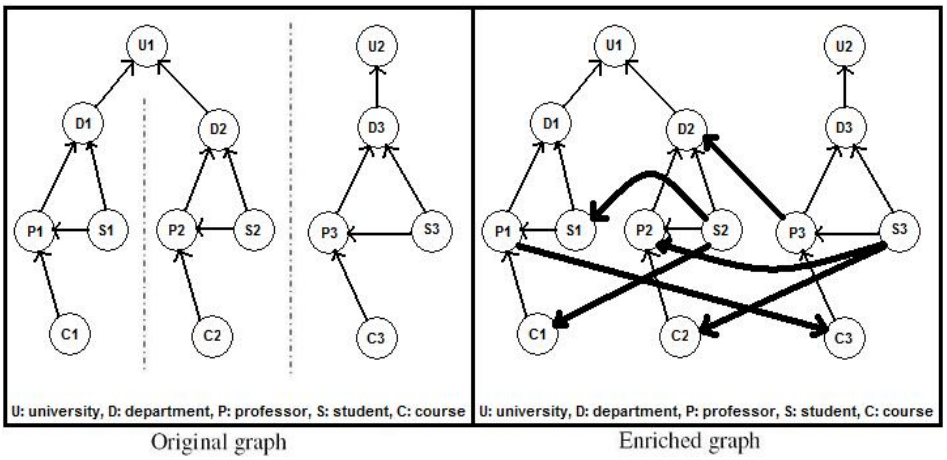

Figure 10: LUBM, ontologies graphs

[21] http://swat.cse.lehigh.edu/projects/lubm/

The advantage with respect to LUBM is that, as we can see in Figure 10, UOBM enriches the ontology produced by LUBM (original graph) by interrelations between Individuals [9].

The disadvantage of UOBM is that it is too difficult for most systems to answer correctly within reasonable time [10].

Other projects have the aim to promote ontologies with a specific domain. They are used in benchmarking because they are huge ontologies.

One of these is GALEN, which consists of a translation of the full Galen ontology (from the original OpenGALEN[22] project) into the OWL description logic. The goal of GALEN is to promote healthcare through collecting experience in this area of study.

Another important ontology is Gene Ontology (GO)[23]. Since 1998 three ontologies have been developed which describe gene products in terms of their associated biological processes, cellular components and molecular functions in a species-independent manner.

Another project, that has the same goal as GO, is SEMINTEC[24]. The SEMINTEC project is an ontology about financial information.

GALEN, GO and SEMINTEC are used in benchmarks through queries which analyze the number of various entities. For example we can check whether the total concepts of ontology calculated by a benchmark is correct. We can do this type of tests for any characteristic of an ontology.

Naturally, when we want to check if the results are correct, we need to know the real number of different entities. Therefore, it is very important that OntoCreator can create huge ontologies from given inputs; this is something of new in the area of semantic applications.

Another reason to use these ontologies is to test the time that the benchmarks need to answer for analyzing huge ontologies.

In summary, current benchmarks must still be improved to get complete coverage of OWL DL and to able to answer correctly within reasonable time.

Regarding OntoCreator, it can contribute in general to the development of the semantic applications area and it can be useful to implement reasoners and benchmarks.

[22] http://www.opengalen.org/
[23] http://www.geneontology.org/
[24] http://www.cs.put.poznan.pl/alawrynowicz/semintec.htm

3 Implementation

This chapter presents a description of the details of OntoCreator, from design phase to the implementation. In the first part are introduced the goal and the requirements of the tool. Then the document describes some possible problems, the architecture and the implementation. Finally an overview of libraries used by the tool is given.

3.1 Goal

In the previous chapters we have seen that, to evaluate semantic applications, it is necessary to have some ontology of different size, depending on the number of entities, and of different complexity, depending on the number of relations. Therefore, it is necessary to develop a tool that is able to generate automatically synthetic OWL1.1[11] ontologies from a set of parameters.

For example, the input of this tool would be the number of Concepts, Properties, and Individuals. Then there are more specifications like Unions, Intersections, Complement, or Cardinality (Exact, Min or Max) and the kind of Properties. Other inputs would be Axioms like Sub-Concepts, Equivalent-Concepts, or Disjoint-Concepts.

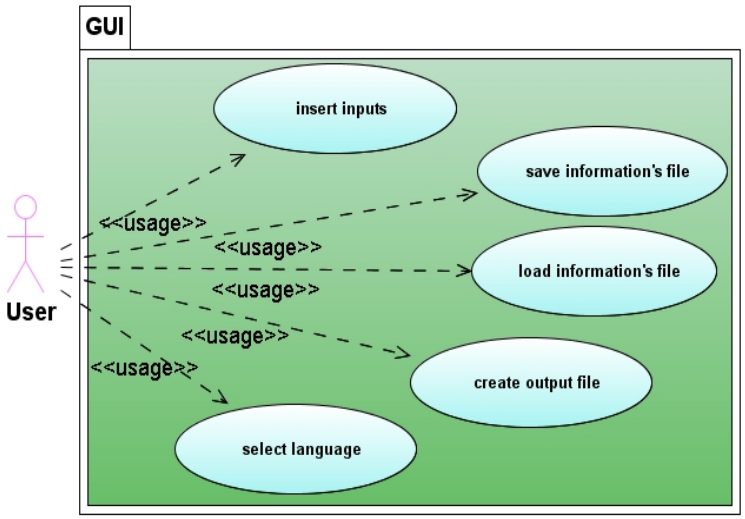

Figure 11: TOOL, use case

As explained in the *use case* of Figure 11, the user can utilize a graphic interface (*GUI*) to set the inputs, to store or to load information, to select the language or to create an output file. The part of the tool, which must take care of all creation's processes, is an *engine*.

Another goal of this thesis is to easily share huge ontologies, without being required to send a file with the size of gigabytes. The problem is that the creation of relations between the entities of the ontology is random, but, to share the ontology, the tool must always produce the same output file from the same inputs. Consequently, we must save all the information (*metadata*) of the creation's process necessary to have the possibility to rebuild the ontology on another computer or in another moment.

The last feature of the tool is that it is multilingual.

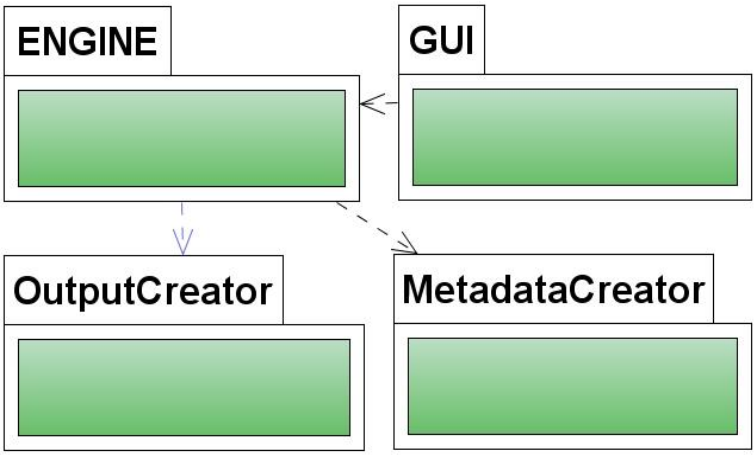

Figure 12: TOOL, packages design

Figure 12 shows the four blocks which belong to the tool and which have just been introduced.

This is generally the scope of the project; it will be further explained in the following sections. In particular, the next session deepens the requirements for the four blocks.

3.2 Requirements

After having understood in previous chapters what is OWL and what is the scope of this thesis, we can now deal with the details of the requirements of this project.

3.2.1 Metadata

The tool must create a text file of metadata, containing the information of the inputs in the GUI.

The metadata file is necessary to share ontologies. In fact, it must be possible to send only this file, which must be around some KB in size, and a link to the tool to share huge ontologies with million of entities, which are usually around some GB in size.

The metadata file must be usually saved with the same name and in the same folder of OWL file. However, the user must be able to specify a different name and output path for the output file by command line.

The metadata file can be created separately from the OWL file. In addition, it must also be possible to load a metadata file through the menu. By this way, all the inputs must be checked before setting all the parameters of the GUI.

3.2.2 GUI

Throught the Graphic User Interface (GUI), the user must be able to save and load metadata file and also to start the OWL1.1 creation.

In the GUI the user can also set the inputs that characterize the ontology. The inputs, which are not the ontology's features, should not be edited by the user.

In addition, the GUI must be multilingual.

3.2.3 Output

The tool must generate OWL1.1 code as a final output. It must always generate the same OWL1.1 code from the same metadata file. This is necessary in order to share ontologies.

The tool must be able to create ontologies with million of entities. This is the most important requirement.

The consistency of the ontology is not a requirement; so, the tool will not guarantee it. Consequently, the engine must not check the consistency of the ontology during the creation of the OWL1.1 file.

The names of instances of Concepts, Properties and Individuals are not important. The only important point is that there is no limit on the number of instantiations. For example, they can be indicated by acronym like C#, P# or I# (where # is a growing number without limits).

3.2.4 Engine

The tool must be implemented in Java (J2SE). It must be usable from a user interface and from command-line.

The tool must allow multiple ontologies are created simultaneously.

It must be possible to easily add other output modules for different final output formats in the future.

The tool must use only external open source libraries.

3.3 Problems

In this section we introduce the difficulties that we need to resolve before starting the implementation.

The most important problems of this thesis concern the limits of memory and the limits imposed by the parser[25].

Memory

The memory in computer applications is managed by heap. A Heap is a quantity of memory assigned to a specific process for storing data structures.

If necessary, the process may require additional heap to the operating system and usually gets double the memory previously allocated. When the memory allocated for a process is unused, the garbage collector can free it.

The JVM has 64 megabytes as default heap, which means that when the process finishes with them it issues an "out of memory" exception. The quantity of default heap is not insignificant, but it is not sufficient to cope with an array with the size of several million objects. So, it is necessary to set the JVM with a certain amount of memory through the option "-Xmx".

Parser

Reasoners need a parser to analyze the syntax of ontologies. In particular, a parser determines the grammatical structure with respect to a given formal grammar. Usually, a parser is a component of an interpreter or a compiler.

OWL1.1 API is the state of art of procedures to manage OWL1.1 ontologies. The OWL1.1 API[26] uses a SAX parser[27] which has a default limitation about the number of entities. This limit is 64000. We need to increase this limit to work with huge ontologies; so, it is necessary to set the parser option relatively the number of entities. It is possible to do this with the command "-DentityExpansionLimit".

[25] In computer science, parsing is the process of analyzing the syntax with a given formal grammar. The most common use of a parser is as a component of a compiler or interpreter.

[26] http://owlapi.sourceforge.net/

[27] http://www.saxproject.org/

3.4 Design

In the previous section, we presented some problems of the tool. Here we introduce different possible choices to determine the design of the tool. The goal of this section is to evaluate advantages and disadvantages of the various possibilities to choose the best solution and to avoid as much as possible to have problems in implementation phase.

Programming Language

The programming language was established by the requirements of the project. In particular, Java language was required. There are several advantages of using Java, for example it is object oriented, it has a greater possibility to reuse code, it is a multi-threaded language, and it is an independent platform. A disadvantage is a slightly lower speed if compared to languages like C and C++.

Metadata

Regarding the metadata file, there are many possibilities to save the inputs; one is to save them in an xml file.

The advantages of xml technology are that it is portable and it includes its significance in the tags. The xml technology is also capable of representing data structures in form of a hierarchical tree. For these reasons xml is designed to exchange data between different platforms. In fact xml documents are text files, so do not require proprietary software to be interpreted.

In this project xml is used only in the OWL1.1 code. For the metadata file we decided that a simple text file to save the inputs is sufficient. This file has the same name and the same path as the OWL file if it is created through the GUI. While, through command line, the user can choose a different name or folder for the file.

The information in the file are stored as a couple of terms representing name and value of the inputs. This is necessary in order to add new inputs in the future. In fact, in this way it will be possible to change the order of inputs in the metadata file without problems.

In Appendix A there is an example of a metadata file; here we can see how the information about inputs are stored.

Engine

Considering OWL1.1 file, the system can use *OWL API* or, having already created Classes that describe Concepts, it can directly use them to generate OWL code.

The problem of OWL API is that it needs to load all the ontology in memory during creation. This is a memory limit to the creation of huge ontologies.

The best solution is to use generic java classes that take care of the output syntax. In this way, it will be possible to easily extend the tool adding other classes in the future. In fact, other classes can represent the semantic with another syntax.

For these reasons we are going to use java classes instead of OWL API.

Command Line

OntoCreator can be launched, along with graphical interface, even from the command line. As explained in Section 3.2.1, the tool must accept some specific options from command line such as the name of the output file.

The tool also needs to receive the inputs relative the ontology's features. Concerning this, we must choose which parameters to insert from the command line.

One solution is to pass all inputs from the command line, but it is inconvenient because in this way, there is a higher risk of writing mistakes and it is not a good engineering solution.

Our choice is that the user must insert the metadata file's path. In this way, the tool can read all the inputs from the file. In addition, this metadata file has the same structure as the file produced with the GUI (in which the order of inputs does not matter). This solution is obviously more expandable and secure.

Concepts Creation

Different ways have been examined for the Concepts creation. The first was to manage all Concepts as Equivalent Axioms.

```
<EquivalentClasses>

  <OWLClass URI="&ns;C500"/>

  <ObjectOneOf>
    <Individual URI="&ns;I455"/>
    <Individual URI="&ns;I128"/>
    <Individual URI="&ns;I173"/>
    <Individual URI="&ns;I775"/>
  </ObjectOneOf>

</EquivalentClasses>
```

In this way, every Class is described by an identifier (*C500* in the example) and it can be used in the following Axioms without repeating the body of the Class, but using only the identifier.

```
<SubClassOf>
  <OWLClass URI="&ns;C500"/>
  <OWLClass URI="&ns;C44"/>
</SubClassOf>
```

This avoids problems of congruence and it allows computational advantage. The disadvantage is that in this way the number of Equivalent Axioms is more than the user inputs expectations.

Another possibility is to use an array. In the array, the tool can directly save the xml code of all Concepts. Otherwise, the system can put in the array objects which represent Concepts.

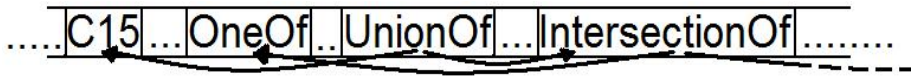

Figure 13: Concepts array with pointers

As Figure 13 shows, the objects can use pointers to reference the other Classes that are members of the Concept.

The advantages of this second solution are a greater reusability and a better implementation through the objects.

There is also an advantage in terms of memory because it does not need to replicate the body of a class in the other cells of the array. The disadvantage is that after having filled the array of Classes and pointers, the tool needs to examine the array another time to resolve dependencies. This requires a higher degree of computational complexity.

Our choice for the project is a combination of the previous two.

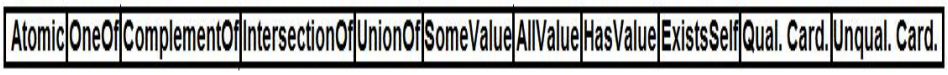

Figure 14: TOOL, Concepts array

As Figure 24 displays, we can uses an array of objects without using pointers, but by copying the body of the Class in the other objects that have it like a member. This can be realized through a recursive function.

Unused Concepts

Concerning unused Concepts, the tool must create the Concepts and it must use them in the Axioms. Only the Axioms are inserted in ontology, but the user requires a total number of Concepts and the system must meet this input. Consequently, the tool must add all the Concepts that are not used in the ontology. Therefore, the tool must store the information about used Concepts during the creation of Axioms and insert the unused Concepts at the end of the process.

A solution for storing this information is to use an array of boolean values. This implementation looks like an optimizing occupation of memory, but we must seek other solutions.

We know that the tool needs to create an array of Concepts. Therefore, another possibility is that the system manages the Concetps information directly into the Concepts array. For example, the first character of each Concept can be equal to "0" if the Concept is unused, while it can be equal to "1" in the other case.

In this way the tool knows which Concepts are unused at the end of Axioms creation. Then, it can insert them in the ontology like

SubClass Axioms of the Class *Thing*.

In this way, a benchmark is able to recognize them because they are the only SubClass of *Thing* that the tool adds in the ontology.

These different implementations will be analyzed in Section 4 with extensive tests to find out which solution involves a greater saving of memory.

Unused Inverse Properties

With unused Inverse Properties, in OWL1.1 there are Inverse Properties defined as Object Property Expressions and Inverse Properties defined as Object Property Axioms. The user wants that the ontology has a certain number of Inverse Properties and that they are used in Property Axioms.

As for the Concepts, the problem is that in the ontology sometimes it is not possible to use all Inverse Properties Expressions in Property Axioms. So, we must add unused Inverse Properties in the ontology; but it is not possible to insert declarations of Inverse Properties Expressions in the ontology.

The solution can be, like for Concepts, to insert the unused Inverse Properties like Property Axioms. But a general entity *Thing* does not exit for the Properties, then we must create one, for example *PThing*.

```
<SubObjectProperties>

  <InverseObjectProperty>
    <ObjectProperty URI="P436"/>
  </InverseObjectProperty>

  <ObjectProperty URI="PThing"/>

</SubObjectProperties>
```

In this way, the tool can insert in an ontology the exact number of Inverse Properties requested by the user and benchmarks can recognize them.

3.5 Architecture

The tool consists of two parts, the first part is a user interface (GUI), that allows the insertion of the inputs. The second part is a core engine that uses the inputs (or the metadata file) to generate OWL1.1 code based on the defined parameters.

The core engine must also be usable from the command line.

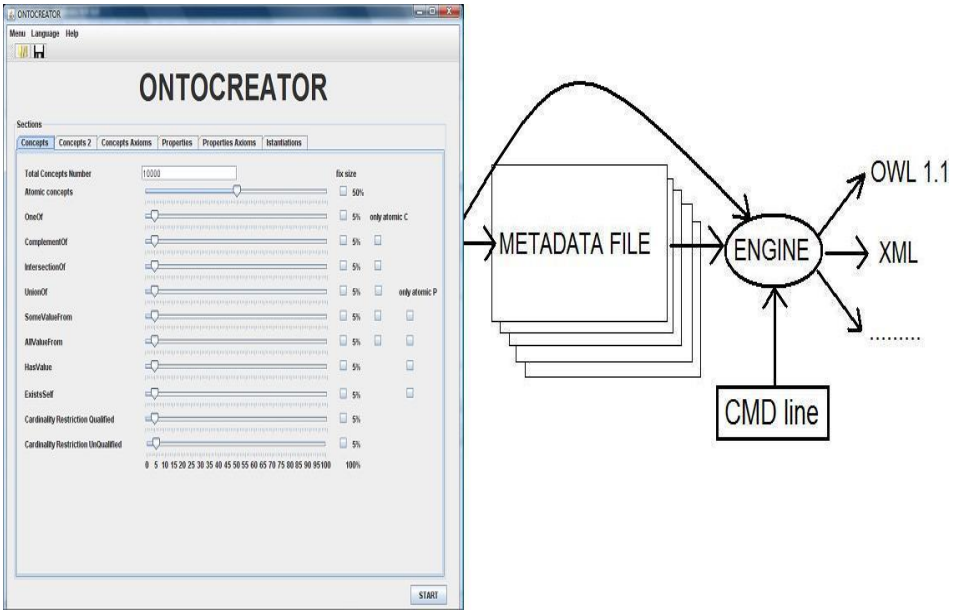

Figure 15: TOOL, architecture

In Figure 15, we can see the presentation layer, which is the user interface, where the user can see the results of a computation, and the processing layer, which is the core engine. This one takes care of all the processes and produces the output files.

Currently the request is to create OWL1.1 file, but in future the tool should be able to create other types of files or other formats that do not exist yet.

3.5.1 GUI

The implementation of a prototype for the GUI has required considerable time. Much work has been done on the structure to create a design divided into semantic groups. In particular it was necessary to decide which inputs to include, which default values to set, which checks to do on the inputs and which engineering features to give to the tool. Some engineering features are, for example, the automatic resizing of the percentages and the checkboxes for the fixing of the value of the respective sliders.

The GUI was divided into six sections. The first two concern the Concepts, with the various percentages and the ranges of Concepts' members. The default ranges' values are set to the numbers that can most often occur. In particular, OneOf, IntersectionOf and UnionOf have a minimum value equal to 2 and a maximum equal to 5. For Exact and Max Cardinalities, the range is equal between 2 and 5.

In the other hand, for Min Cardinalities, the range is between 0 and 2. It was decided voluntarily to set this minimum value to 0 because it highlights the fact that it is possible to have Min Cardinality with a value equal to zero.

The next Section considers the Concepts Axioms. After this, there are the Properties and the Properties Axioms. According to OWL semantic, with the Properties Axioms there are also the Range and Domain Axioms.

In the last section there are Individuals, the Individuals Axioms, the Assertions and the information necessary to build an ontology. These inputs were grouped together for a matter of space.

In Figure 15, we can see that the GUI presents three menus. In the first menu there is the possibility of creating the metadata file (item "Save MetaDataFile"), specifying the name of the file. This function is also accessible by a button under the menu. Another possibility is to retrieve a metadata file (item "Load MetaDataFile").

The second menu concerns the languages, where the user can choose the language of the labels used by the GUI. At the present, the user can choose between English, German and Italian.

The last menu contains only helpful information like the documentation and the thesis details.

With the button "START" the user can start the output creation process.

Concepts I

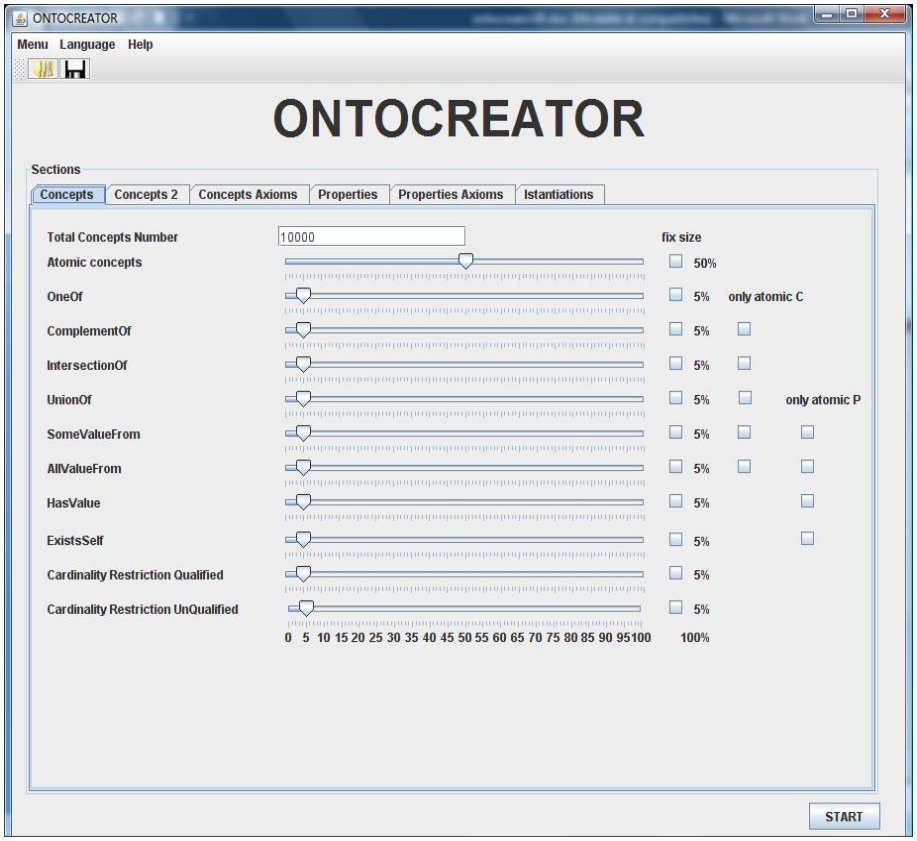

Figure 16: TOOL, Section concepts I

The inputs concerning Concepts are split in two forms. As we can see in Figure 16, in the first there are the total number of Concepts and the percentages of various types of Concepts. Each percentage refers to the total number of Concepts and defines the number of different kinds of Concepts that must be present in the ontology.

For Complements, Intersections, Unions, SameValue and AllValue Concepts, the user can decide if they are combinations of only Atomic Concepts. While, for SameValue, AllValue, HasValue and ExistsSelf Concepts, the user can decide if they are combinations of Atomic Properties.

Concepts II

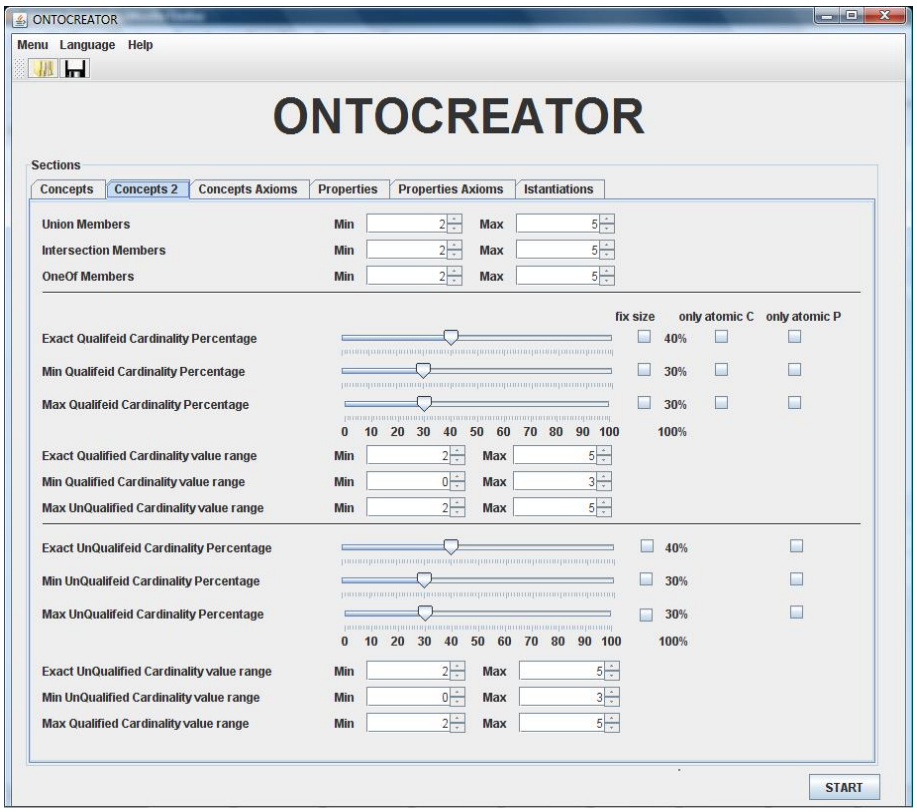

Figure 17: TOOL, section concepts II

In Figure 17 there is the second section concerning the Concepts. Here, the user must insert a minimum and a maximum number of members for Unions, Intersections and OneOf Concepts. After there are two subsections for Qualified and Unqualified Cardinality restrictions. For the Cardinality restrictions the user must specify a range between the minimum and the maximum values.

The Qualified and the Unqualified Cardinality percentages are split into three different types: Min, Max and Exact Cardinality. The user can decide with all types of Qualified Cardinality restrictions whether they are combinations of Atomic Concepts and Atomic Properties. For Unqualified Cardinalities there is not the "only Atomic

Concepts" checkbox because the Range is the Class *Thing*.

If the user sets the percentage of a Concept kind to zero in the previous section of the Concepts, the details of this Concept are hereby disabled.

Concept Axioms

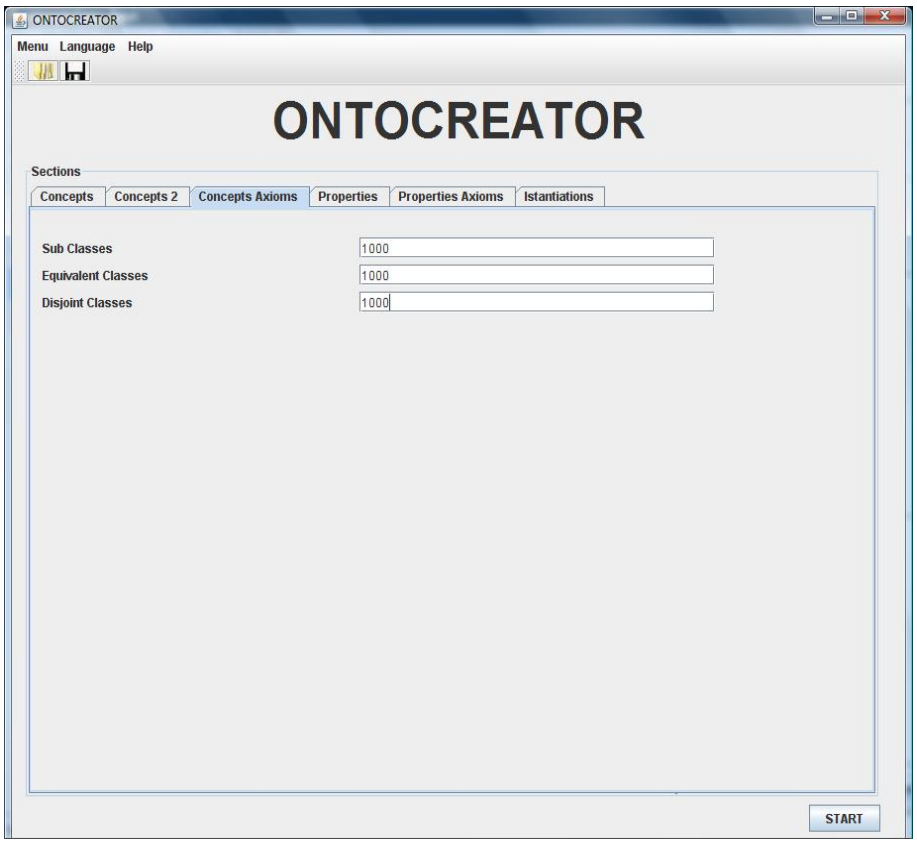

Figure 18: TOOL, section concept Axioms

In Figure 18 there is the section of the GUI concerning the Concept Axioms. Here the user can insert the numbers of sub, equivalent and disjoint Axioms.

To emphasize the difference between Axioms and the entity's descriptions, the user must define a number for the different types of

Axioms that must be present in the ontology. In the other side, for entity's descriptions he shall set the percentage respect to the total number of Concepts.

Properties

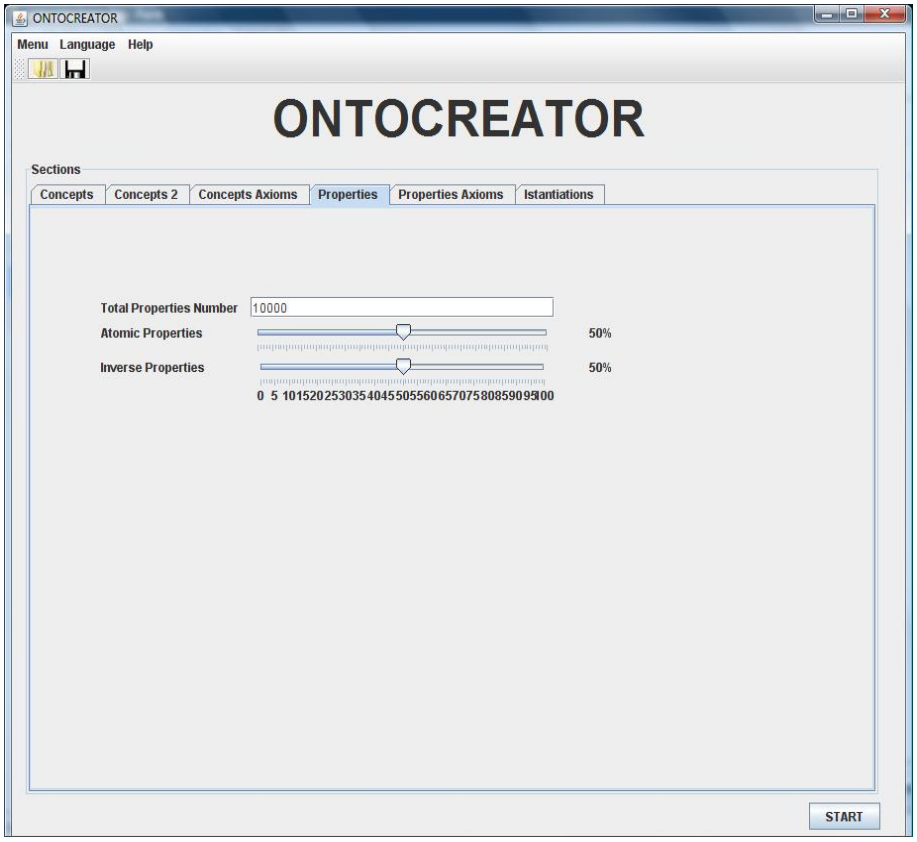

Figure 19: TOOL, section Properties

In the section concerning Properties (Figure 19), the user can insert the total number of Properties. The Properties are divided into Atomic and Inverse types.

Inverse Properties can refer to Atomic Properties or to Inverse Properties. So, it is possible to have Inverse Properties of Inverse Property.

Properties Axioms

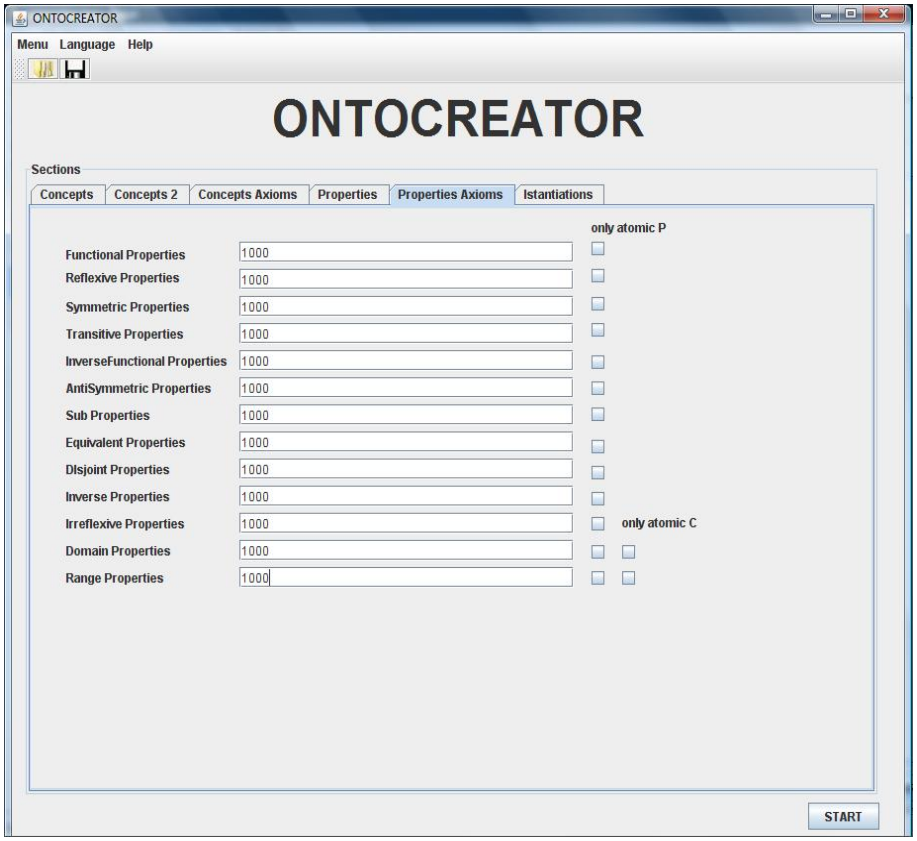

Figure 20: TOOL, section Properties Axioms

In Figure 20 there are all types of Property Axioms.Here, the user can specify if they are combinations of only Atomic Properties. The user can also specify if they use only Atomic concepts for Domain and Range.

Every number of Properties Axioms, except for Sub, Equivalent and Disjoint Properties Axioms, must be less than the total number of Properties. If the Axiom uses only Atomic Properties the number must be less than the number of Atomic Properties.

Individuals

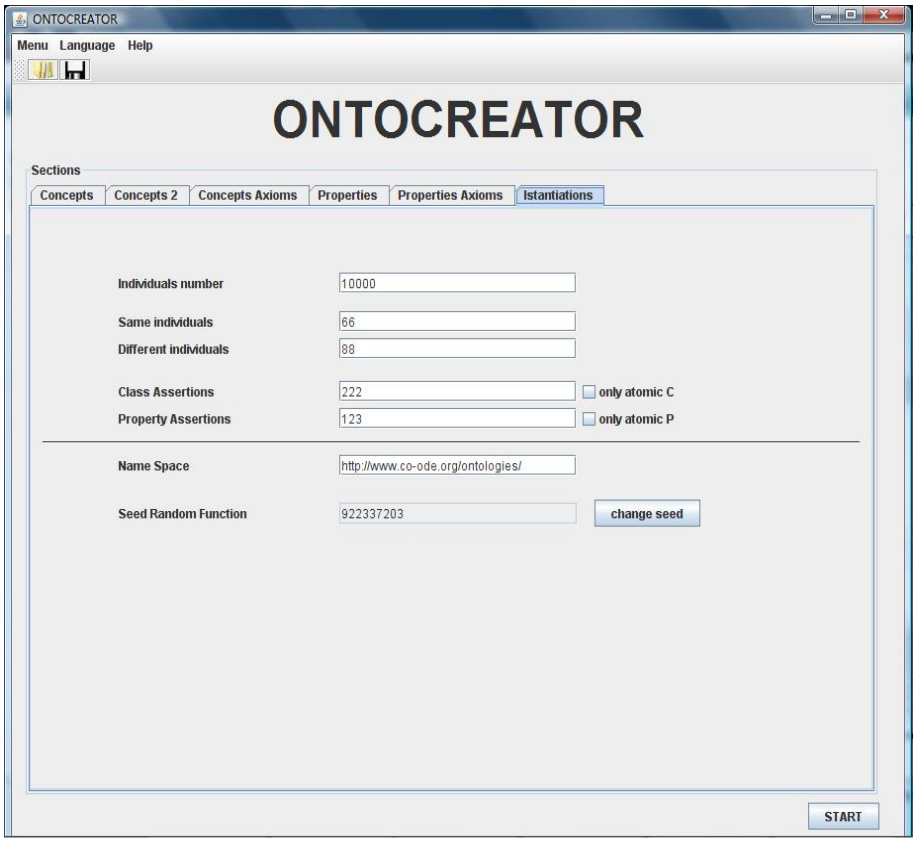

Figure 21: TOOL, section Individuals

In the last section of the GUI (Figure 21), the user can insert the inputs on Individuals and Assertions.

Here there are the total number of Individuals, the number of Same Individuals and the number of Different Individuals.

The number of total Individuals must be greater than zero. Onthe-contrary, there are no checks on the size for the numbers of Same and Different Individuals, but the tool always tries to insert a different couple of Individuals.

Then there is the number of Class Assertions. A Class Assertion specifies which Individuals belong to which Classes. If the checkbox *only Atomic Concept* is active, the Individuals can belong only to Atomic Concepts.

After that, there is the number of *Property Assertions*. A Property Assertion specifies that one Individual is connected with another Individual (or the same) by a Property. If the checkbox *only Atomic Property* is active, the Individuals can only be connected by Atomic Properties.

Ontology Information

In the Individual's section there are also, for design reasons, the ontology information.

In fact, there is the *Name Space*, that is the name path of Classes, Properties and Individuals. It can be of http type (for example *http://www.co.ode.org/ontologies/*) or of file type (for example *file:/C:/*).

The other input is the seed of the random function. As was requested by requirements, the user can not insert this but he can change it by a button.

Progress bar

The user may need to verify that the tool is working on ontology creation, especially if he wants to create multiple ontologies simultaneously.

In order to keep track of active threads and of their progress, the system uses a separate window to show the most important information of the ontology and the progress bar of the thread.

One possibility could be to update the progress bar on the basis of an estimated time, but this would be dependent on the machine where the tool is running, on the allocated resources and on the other active processes.

The progress bar updates its state at the end of different steps.

In particular, as we can see in Figure 22, there are 11 steps. These steps will be further explained in Section 3.6. The end of the first eleven steps increase the progress bar by 9%. A further last step, concerning unused Concepts, adds the last 1%. This approximation is congruent with the real time spent in these steps.

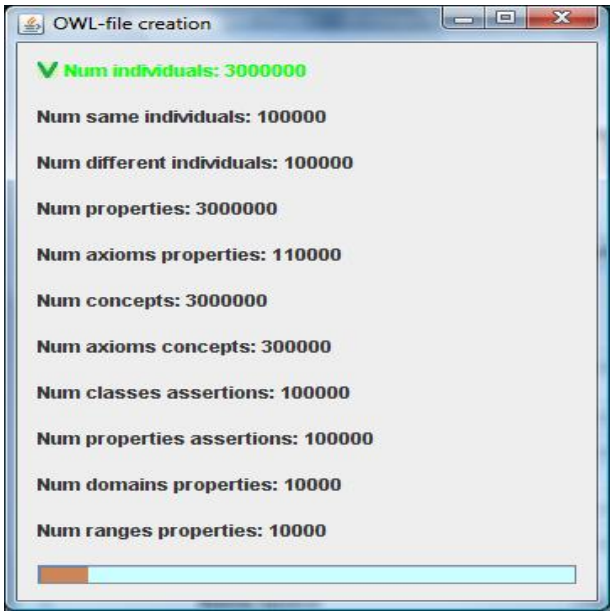

Figure 22: TOOL, progress bar

3.5.2 Core engine

As was explained in requirements, the tool must concurrently create more ontologies. So, the user can start the OWL creation and then work over the GUI to create another ontology.

A solution can be to have a thread that manages the whole process. The problem is that in this way the system can create only one ontology at a time. Instead, a good solution can be to create one thread for every request of OWL creation. Therefore, the user can launch more processes at one time.

3.6 Implementation

This section explains the operations that the engine carries out to create the outputs. The OntoCreator process manages the creation of metadata file and of OWL file.

In the implementation the tool creates before the metadata file and after the output file. The reason for this is because if the system crashes during the OWL code's creation it is possible to recover the inputs in the GUI.

Concerning the creation of the metadata file, the tool writes the inputs into a text file like couples of input name and input value.

Regarding the OWL creation process, it is split into 12 steps.

1. The tool starts writing the Individuals because they are the simpler part of an OntoCreator process due to lack of relations with Concepts and Properties.

2. After the tool inserts the Same Individuals in the ontology. Each Individual is defined by a number that is in the range of zero to the total number of Individuals. So, the tool chooses two random numbers to define in an Axiom that two Individuals are equal. It is important to emphasize that the tool can add that an Individual is equal to himself.

3. The next step concerns the Different Individuals creation. The system monitors so as not to insert an Individual defined as different from itself in an Ontology because this type of relation may often occur and there is a high risk of inconsistent ontology. This is the only check that the tool does before writing them in the output file because the consistency is not sought.

4. After Individuals, the tool generates all Atomic and Inverse Properties. Atomic Properties, like Individuals, are written in the ontology like a declaration. This is necessary because they are not Axioms.

5. Then it decides randomly which Properties are from a specific kind of Property Axiom. A Property can be, as just explained in Section 2.2.2, Inverse, Functional, Inverse-functional, Reflexive, Irreflexive, Symmetric, Antisymmetric or Transitive. A Property

can belong to different types, also to all types; in fact, any combination is possible. Then, the system creates the Sub, Equivalent and Disjoint Property Axioms. Every Axiom uses two different random Properties.

6. In the next step, the tool creates and stores in the memory an array of Concepts that has a size equal to the total number of Concepts.

   According to various percentages required for the different types of Concepts and as explained by Figure 23, in the array they are in the following order:

   - Atomic Concepts,
   - OneOf Concepts,
   - ComplementOf Concepts,
   - IntersectionOf Concepts,
   - UnionOf Concepts,
   - SomeValue Concepts,
   - AllValue Concepts,
   - HasValue Concepts,
   - ExistsSelf Concepts,
   - Exact, Min and Max Qualified Cardinality,
   - Exact, Min and Max Unqualified Cardinality.

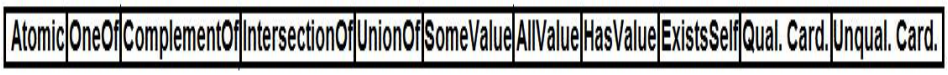

Figure 23: TOOL, Concepts array

The array contains the specific implementations of objects that represent the different kinds of Concepts. Every object implements the java class *ConceptOWL* and specifics its own method of getting the xml code.

In the first part of the Concepts creation the tool generates the Atomic and OneOf Concepts because they are the easiest components since they have no references to the other Concepts.

The tool stores every Concept in the respective cell of the array, e.g. Class 1 is in cell 1, etc.

Then, it creates the ComplementOf Concepts generating a random number in the range of the total number of Concepts. This number represents the member of the ComplementOf Concept. The tool needs it to determine the reference to the other Concept. With this number it checks that the ComplementOf Concept does not have a reference to itself because this would imply an empty universe.

The tool starts to create the array from the first cell to the last, i.e. from the first Atomic Concepts to the last Maximum Unqualified Cardinality Concepts. So, during the process, the tool know that if the generated random number is less than the index of the current object creation, the referred concept must necessarily have just been created. In this case, the ComplementOf Concept can directly copy the xml code inside. Otherwise, the tool must check whether the Concept has already been created, and if it has not, it must create and save it in the correct place in the array. The creation of the Concept means that if it needs references to other Classes, the system must create them with the same procedure.

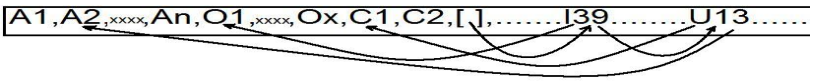

Figure 24: TOOL, Concepts array creation

In Figure 24 an example is shown to understand what happens if a member of Concept has not yet been created. In the example the tool has already stored all the Atomic and OneOf Concepts. Also the first two ComplementOf Concepts have already been created and they have just copied the xml code of their respective member. The tool is now creating the third ComplementOf Concept and it requires the IntersectionOf Concept *I39* as his member. Concept *I39* has not already been created; so the tool must ensure this. In particular, *I39* has two members; one is the OneOf Concept *I39* that has already been created. The other one is the UnionOf Concept *U13*. The tool checks recursively for

each member if it has already been created in the memory and in the negative case it provides for the generation of its members and copies the xml code in its body. In the example, the Class *U13* is the union of the Class *A2* and the Class *C1*. So, the recursion can stop now and the ComplementOf Concept *C3* can be stored in the memory.

This recursive function can potentially lead to a stack overflow, but for statistical probability and for the presence of Atomic and OneOf Concepts this has never happened in tests that have been carried out.

For IntersectionOf and UnionOf Concepts, the tool generates a random number in the range between the minimum number of members and the maximum number of members for each Concept to determine the number of members. After this, the Concept obtains the xml code from each member.

The system must generate a Property for SomeValue, AllValue, HasValue, ExistsSelf and Cardinality Concepts. So, the tool decides a random number in the range of the total number of Properties and it inserts this Property in the xml code of the Concept.

The tool also generates an Individual with a random number in the range of the total number of Individuals for HasValue Concepts.

For the Cardinality restrictions, the system generates the value of the cardinality for each Concept.

7. Then, the tool creates Sub, Equivalent and Disjoints Concepts Axioms using two different random Concepts, like for the Property Axioms. In contrast to Property Axioms, in the case of Concept Axioms the tool uses the array to obtain the two Classes. The Axioms are directly written in the output file.

8. After this, it randomly defines Class Assertions. For a Class Assertion the tool decides that a random Individual belongs to a random Class.

9. The next step is the creation and the writing in the output file of Property Assertions.

   It decides for Property Assertions that a random Individual is connected by a random Property to another (or the same) random Individual.

10. The tool now creates the Domain Axioms. It must choose at random a Property and a Class. We must remember that Domain Axioms are characterized by the checkboxes "only Atomic Concepts" and "only Atomic Properties" simultaneously.

11. For the Range Axioms the process is similar to that of Domain Axioms.

    During the creation of the Concept Axioms, Class Assertion, Domain Axioms and Range Axioms the tool stores the information about the using of the Concept in the java class that represents the object.

12. Finally, the tool explores every Concept in the array to check if it was used in the Axioms. In the negative case, the unused Concept is added in the ontology like SubClass of the Class Thing.

If a typology of Concept has the checkbox *only Atomic Concepts* active, the references are only to the pool of Atomic Concepts, which are in the first part of array. The same idea is applicable for checkbox *only Atomic Property*; if it is active the class will only use Atomic Property.

It is important to emphasize again that the ontology may or may not be consistent because it is not a requirement.

In Appendix B there is an example of an ontology with only few Concepts, Properties, Individuals and Axioms. As we can see, although its size is much smaller than an ontology of millions of entities, the size of text files is just significant.

3.6.1 UML

This section presents the UML[12] representation of the tool implementation. The Unified Modeling Language (UML), an Object Management Groups (OMG) project, is an industry "de facto" standard for modeling software. For this reason, generating code from UML models is desirable.

UML is a graphical modeling language used for specifying, visualizing, constructing, and documenting software intensive artifacts. Generation of code from class diagrams is supported by most IDE tools.

With UML diagrams it is possible to have an efficient understanding of the system's modules without having to read the code. In engineering process, the systems code is specified and constructed from the UML models. Moreover, UML has the power to hide unnecessary details of the system with the ability to model its different views.

UML helped to build a well-structured object-oriented application, regardless of programming language that will be subsequently used. When the UML models of a system are successfully built, we can map them to a source code of some object-oriented programming languages, such as C++, Java, Visual Basic, or tables in a relational database. Mapping the models into a code is known as *forward engineering*.

Forward engineering is not applicable to all UML graphs, but it depends on language and on the level of abstraction of the model. However, it is always possible to map the class diagrams in Java classes.

In this project, UML was used to create the class diagrams for Concepts, Properties, Individuals and Axioms.

We can start to analyze the UML graphs of the tool from the class diagram for the Concepts because it has the greater expressiveness.

This diagram is too big to be shown on only one page, so it is split in Figure 25 and 26.

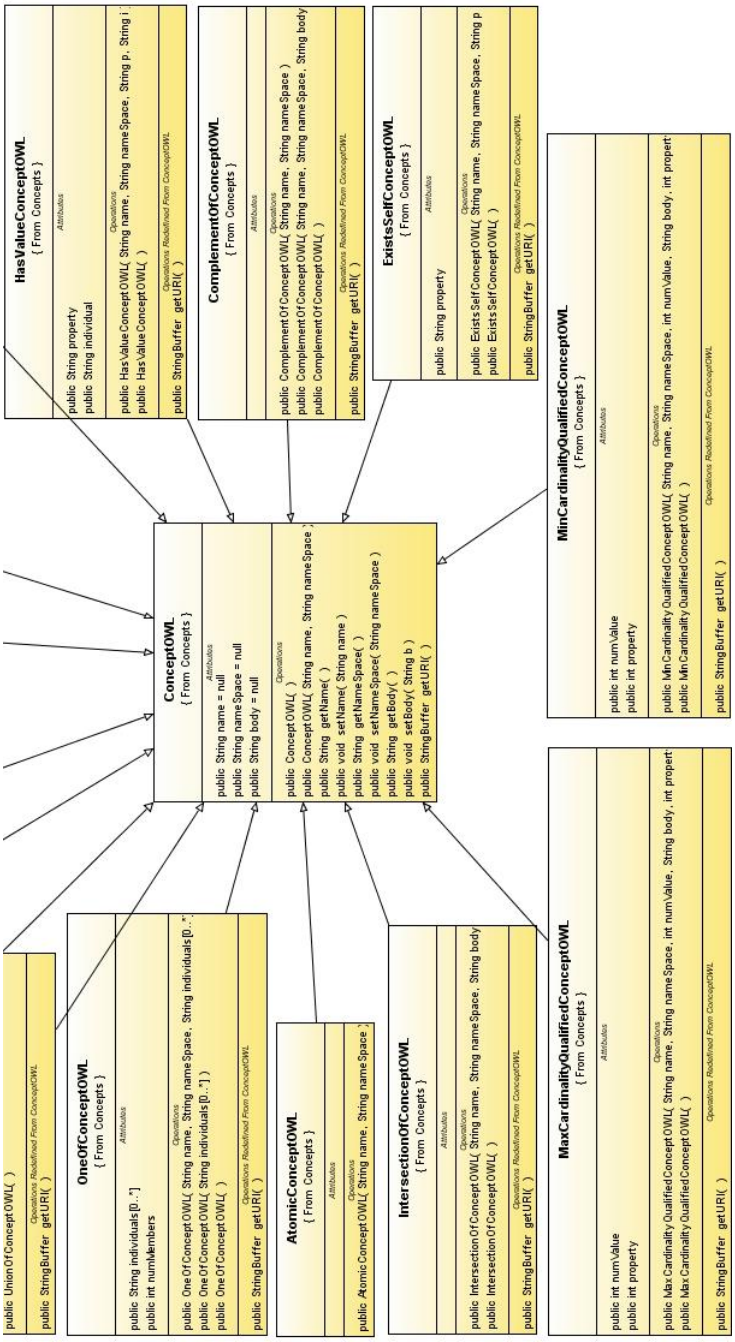

Figure 25: UML, Concepts classes diagram I

Figure 26: UML, Concepts classes diagram II

In Figure 25 and 26 it is shown that all types of Concepts implement the class *ConceptOWL*.

A feature of all Concepts is that, except the Atomic Concepts, they override the method *getURI* of ConceptOWL. In fact, every Concept has its own particular syntax. All Concepts are characterized by a *name*, by a *name space* and by a *body*. The body is the xml code for the members of the Concept. For Atomic, HasValue, ExistsSelf and Unqualified Cardinality Concepts, the body is void because these concepts have no members. HasValue and OneOf Concepts have a reference to Individuals. While SomeValue, AllValue, HasValue, ExistsSelf and Cardinality Concepts have a reference to a Property.

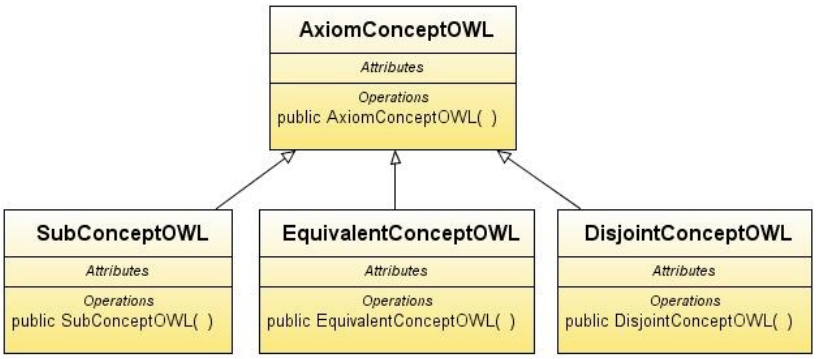

Figure 27: UML, Concept Axioms classes diagram

Figure 27 shows the UML relations concerning Concept Axioms. As we can see, there are the three kinds of Concept Axioms here, i.e. Sub, Equivalent and Disjoint Concept Axioms.

In Figure 28 there is the class diagram of Properties. Here there are only two possible descriptions of Properties: Atomic and Inverse Property.

Property, as Concept, is characterized by a *name*, by a *name space* and by a *body*.

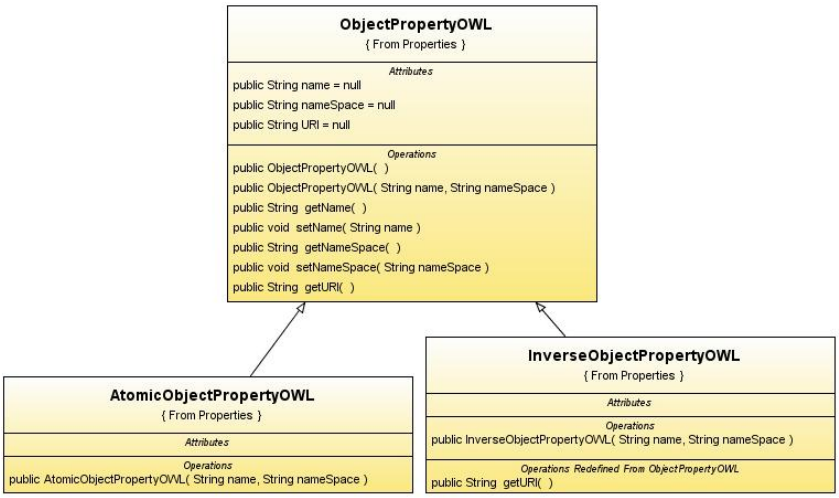

Figure 28: UML, Properties classes diagram

As explained in section 3.4, there is a difference between Inverse Properties defined as Object Property Expressions and Inverse Properties defined as Object Property Axioms.

For example, an Inverse Property Axiom may contain two Inverse Properties, such as:

```
<InverseObjectProperties>

    <InverseObjectProperty>
        <ObjectProperty URI="P77"/>
    </InverseObjectProperty>

    <InverseObjectProperty>
        <ObjectProperty URI="P76"/>
    </InverseObjectProperty>

</InverseObjectProperties>
```

While an unused Inverse Property is added to the ontology like an Sub Properties Axiom. This is because ontologies can not contain declaration of Inverse Properties Expressions, but it can only contain Inverse Properties in Properties Axioms.

An example of unused Inverse Property is:

```
<SubObjectProperties>
    <InverseObjectProperty>
          <ObjectProperty URI="P436"/>
    </InverseObjectProperty>
    <ObjectProperty URI="PThing"/>
</SubObjectProperties>
```

This features of Properties Axioms are shown in the class diagram of Figure 29.

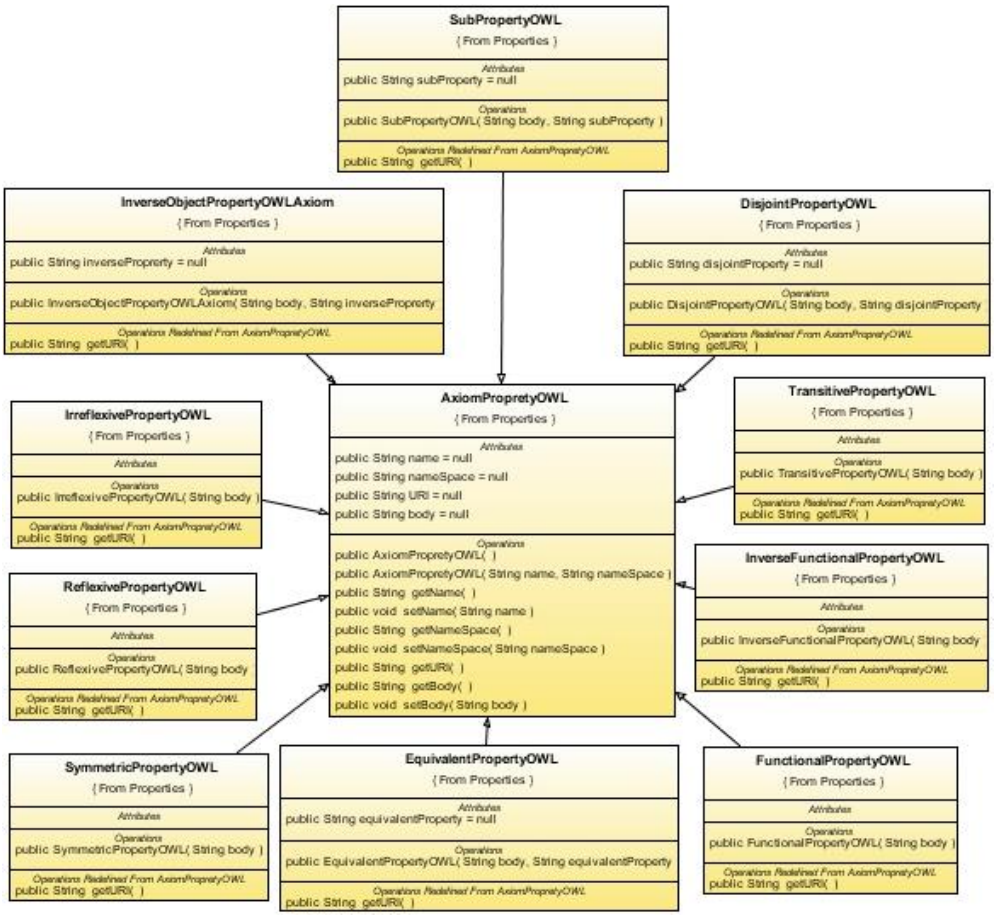

Figure 29: UML, Properties Axioms classes diagram

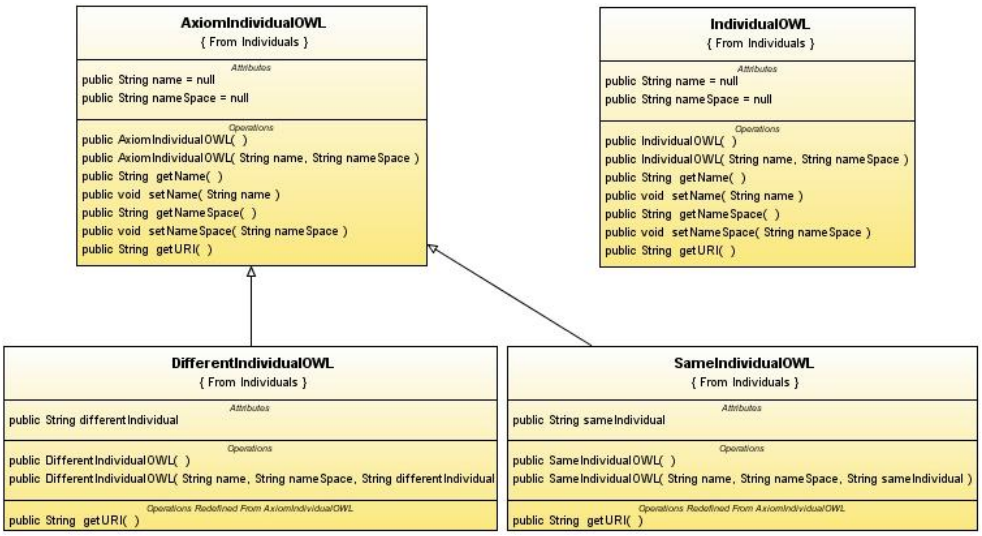

Figure 30: UML, Individuals classes diagram

Finally, in Figure 30 there is the class diagram of Individuals. Here, we can see that Same Individuals and Different Individuals are Individual Axioms. They have the same structure of Concept and Property Axioms.

In this section we have seen the main features of tool's implementation. Therefore, we have been able to deepen our understanding of the system's structure without having to read the code.

3.6.2 Java Web Start

The final phase of this thesis requires the online publication of OntoCreator.

In this way, the user can share huge ontologies sending only the metadata file, which is small in size, and the link of the tool's web page.

Java Web Start[28] (JWS) was requested to avoid problems of versioning and uploading.

The technology JWS was presented for the first time in San Francisco at the JavaONE in the year 2000; it was created as an alternative to the *applet*.

An applet is a software component that runs in the context of another program, for example a web browser. Its disadvantage is that we can use it only if we are connected to the internet.

With Java We Start, it is possible to download and use a standalone remote application without the web browser. The most important advantage is that JWS takes care of the versioning of the software, i.e. it provides an automatic updating.

One interesting feature of JWS is that the application is locally installed. In fact, as a result of this, it is possible to have an icon on the desktop (or an item on the Start menu) that allows for the launching of the application. The exception being the first time when we need to launch the application through a link to a JNLP file in a web page.

Java Web Start uses JNLP (Java Network Launching Protocol) to allow the download of the application from a server and to manage the uploading of new versions. JNLP consists of a set of rules defining how exactly this launching mechanism should be implemented. JNLP files include information such as the location of the jar package file and the name of the main class for the application.

With a properly configured browser, JNLP files are passed to a Java Runtime Environment which in turn downloads the application onto the user's machine and starts executing it.

Another important feature of JWS is that it is free, so developers are not required to pay a license fee in order to use it in programs.

[28] http://java.sun.com/products/javawebstart/

Figure 31: TOOL, Java Web Start

An example of JNLP file is:

```
<jnlp spec="1.0+"
 codebase="http://localhost:8080/OntoCreator"
                      href=" OntoCreator.jnlp">
  <information>
    <title>OntoCreator</title>
    <offline-allowed/>
  </information>
  <security>
    <all-permissions/>
  </security>
  <resources>
    <j2se version="1.4+" max-heap-size="512m" />
    <jar href=" OntoCreator.jar"/>
  </resources>
  <application-desc main-class="src.Main" />
</jnlp>
```

The parameter *codebase* sets the server information with the location of the file jar; *offline-allowed* indicates that we can launch the file JAR even if we are not online. *All-permissions* means that we do not need to be an administrator to launch the application. In *j2se* there are the options for the JVM.

The JNLP file is downloaded by the user from a server and it is necessary to start the application the first time.

With JWS, we can automate the versioning and the uploading of the tool. In this way, when we share ontologies, we are sure that the OWL files created with the tool by a metadata file are always the same.

3.6.3 Libraries

This section deals with the libraries used in the thesis. A library is a set of APIs with a particular purpose.

As was explained in Chapter 3.2.4, a goal of this project is to use only open source libraries. The reason for this choice is to limit costs and to have an easy distribution of the software.

The following are the three libraries mentioned that represent the solution to the problems previously seen.

Swing Layout Extensions

Swing Layout Extensions[29] library simplifies the creation of professional layouts with Swing. This library was developed as an open source project by *Java.net*. The Swing Layout Extensions library is a set of classes extending the layout capabilities of Swing. The main features are automatic aligning on baseline, platform independent spacing and new layout managers.

JARGS

JArgs[30] provides a convenient and compact suite of command line options. The advantages of JArgs are that it is easy to use, well documented and open source. Another advantage of JArgs is that the package is small (only 200KB).

OWL1.1 API

OWL1.1 API[31] is a Java implementation of OWL. It is compatible with OWL-Lite, OWL-DL and it has some elements of OWL-Full. This library is open source and available under the LGPL License[32].

The OWL API includes an in-memory reference implementation, a RDF parser and a OWL parser.

This library was used to test the performances of the tool and to analyze the results. It is not included in the last version of the tool.

[29] https://swing-layout.dev.java.net/
[30] http://jargs.sourceforge.net
[31] http://owlapi.sourceforge.net/
[32] The GNU Library General Public License (or LGPL) is a free software license.

4 System Evaluation

This chapter evaluates the performances and the output files of OntoCreator.

First, we investigate the correctness of the tool. Then, we measure the performances and overheads of OntoCreator to check the limits of the project. The goal is to improve and to perform the tool. Finally, we are going to estimate the consistency of ontologies to understand the relations between the inputs and the output files.

4.1 Analysis

This section presents the analysis of the output files. Here we want to analyze the ontologies producted by the tool. In particular we are going to analyze the correctness, the congruence between inputs and numbers of entities and the percentages of referenced entities.

We expect that the number of referenced instances are directly proportional to the number of Axioms. We also expect that to reference all the instances of Concepts, Properties and Individuals is necessary a number of Axioms greater than the number of instances. This is because the Axioms can use the same entities several times.

The tests were performed with *OWL1.1 API*. Some clarifications on how the ontology manager of OWL1.1 API counts the different instances are necessary to understand the results of the tests.

Concerning the number of Same Individual instances, the system also counts Individuals that are equal to themselves.

If a Class X equivalent (or disjoint) to a Class Y is defined in the ontology, then the ontology's manager does not count the definition of the Class Y equivalent (or disjoint) to the Class X.

If in the ontology it is defined a Class X like Sub Class of a Class Y and there is also the definition of a Class Y like Sub Class of a Class X, the ontology's manager counts both the instances because they are different.

During the creation of the ontology, the tool tries to avoid the writing of identical Axioms. To do this, the tool checks the entities using only the numbers of the instances. For example, it writes only one time that the Concept 98 is disjoint to the Concept 134, but there are no checks on the possibility that the semantic of two different

Classes is the same. For example, the semantic of the Union of two Equivalent Classes is equal to the two single Classes.

All the tests are a series of twenty repetitions on command line. The repetitions were no more then twenty because these tests are stressful and they require a considerable amount of time.

The tests were performed by an automatic suite. It creates the metadata file, then it creates the OWL1.1 file from the metadata file and finally it checks the ontology. The metadata file is always different in every repetition.

The first test was to create an ontology with 10.000 Concepts (50% Atomic, 5% for each other kind), 10.000 Properties (50% Atomic, 50% inverse), 10.000 Individuals and without Axioms.

The results of the test are:

Unused Concepts:	9986
Properties Referenced:	6732
Individuals Referenced:	1991
Axioms:	0

As expected, there are no Axioms in the results. One interesting information is that the total number of unused Concepts recognized by the tests is 9986, approx 100% of the total number of Concepts. This means that there are 14 identical Concepts. This is a result that we expected from a ontology without Axioms and where the unused Concepts are included.

Another interesting point is the number of referenced Properties. It is equal to 6732, approx 67% of the total number of Properties. In this case, the reason is that Properties are only used in unused Inverse Property (50%) and in SomeValue, AllValue, HasValue, ExistsSelf, Qualified and Unqualified Cardinality Concepts (30%, but about half can be an unused Inverse Properties).

The last parameter is the number of referenced Individuals, which is equal to 1991, approx 20% of the total number of Individuals. The reason for this amount is that they are only used in OneOf (5% for an average of three instances) and HasValue Concepts (5%).

The next tests were performed with the same number of Concepts, Properties and Individuals, but with 1000 concept Axioms, 1000 Properties Axioms and 1000 Individuals Axioms used, i.e. 10% of the total.

The results of the test are:

SubClassOf Axioms:	100
EquivalentClasses Axioms:	100
DisjointClasses Axioms:	100
Unused Concepts:	9043
ClassAssertion:	100
Properties Referenced:	7120
ObjectPropertyAssertion:	100
FunctionalObjectProperty:	50
ReflexiveObjectProperty:	50
TransitiveObjectProperty:	50
Sym Properties:	50
Inverse Properties:	50
InverseFunctionalObjectProperty:	50
AntiSymmetricObjectProperty:	50
SubObjectProperty:	33
Equi Properties:	33
DisjointObjectProperties:	33
IrreflexiveObjectProperty:	50
ObjectPropertyDomain:	150
ObjectPropertyRange:	150
Individuals Referenced:	2753
SameIndividual:	175
DifferentIndividuals:	175

Correctly, all the numbers of Axioms are congruent with the inputs of the tests.

As expected, the numbers of referenced Properties and Individuals, respectively 7120 and 2753, are slightly higher when compared with the previous tests. Obviously, the number of unused Concepts is slightly decreased and it is equal to 9043.

So, for now, the tests confirm what we had assumed at the beginning of this Section.

The next tests were done with the double the amount of Axioms compared to the previous tests.

The results of the test are:

SubClassOf Axioms:	200
EquivalentClasses Axioms:	200
DisjointClasses Axioms:	200
SubClassOf *Thing* Axioms:	8242
ClassAssertion:	200
Properties Referenced:	7414
ObjectPropertyAssertion:	200
FunctionalObjectProperty:	100
ReflexiveObjectProperty:	100
TransitiveObjectProperty:	100
Sym Properties:	100
Inverse Properties:	100
InverseFunctionalObjectProperty:	100
AntiSymmetricObjectProperty:	100
SubObjectProperty:	66
Equi Properties:	66
DisjointObjectProperties:	66
IrreflexiveObjectProperty:	100
ObjectPropertyDomain:	300
ObjectPropertyRange:	300
Individuals Referenced:	3464
SameIndividual:	350
DifferentIndividuals:	350

Again, the numbers of Axioms are congruent with the inputs. The total number of unused Concepts continues to decline, while the numbers of Properties and Individuals continue to increase.

The results of all the tests are shown in a compact way in Tables 7 and 8.

Table 7: Number of unused Concepts for different numbers of Axioms

Axioms	0	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
unused C.	9986	9043	8242	7432	6774	6149	5573	5058	4605	4110	3660

Table 8: Referenced Properties and Individuals for different numbers of Axioms

Concepts	0	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	20000	40000
Properties	6732	7120	7414	7658	7846	8085	8276	8452	8637	8750	8850	9660	9980
Individuals	1991	2753	3464	4096	4638	5157	5583	6003	6381	6770	7140	8800	9850

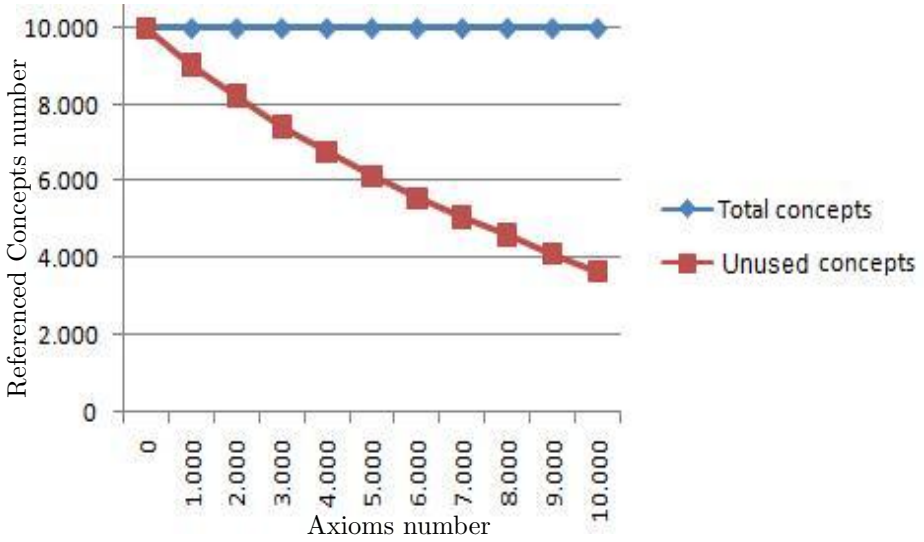

Figure 32: TEST, analysis of concepts

Figure 32 shows the data of Table 7 as a graph. The horizontal axis represents the number Axioms. The vertical axis represents the number of referenced Concepts.

As expected, we can see that, if there are no Axioms, no Concept is used. Obviously, this number decreases when increasing the number of Axioms. So, the number of unused Concepts is inversely proportional to the number of Axioms. The trend, as we expected, is not linear because if the number of Axioms increases, the probability that two Axioms use the same Concept also increases.

Figure 33 displays the graph for Table 8. We have here the numbers of referenced Properties and Individuals on the vertical axis. In the horizontal axis there are, like before, the numbers of Axioms.

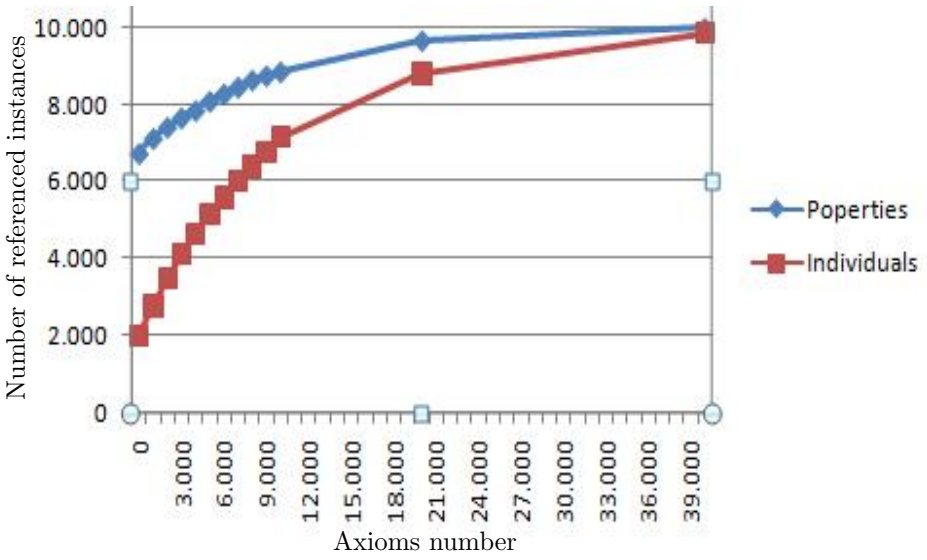

Figure 33: TEST, analysis of Properties and Individuals

As we expect, the values of referenced Properties and Individuals goes up with the increasing of Axioms number. It is important to note that all the instances are referenced when the number of Axioms is around four times more than the total number of entities.

We can also note that, if there are no Axioms in ontologies, the percentage of Individuals is near 20%. As we just explained, the reason for this amount is that Individuals are used in OneOf Concepts (5% of 10000 with an average of three members each instance) and in HasValue Concepts (another 5%).

Regarding Properties, there are about 67% of referenced Properties without Axioms. This occurs because around 50% of the Properties are added like unused Inverse Properties. The rest is from the other Concepts which have a reference to a Property.

Another important characteristic, which has emerged from the tests, is that the values of referenced instances are quite independent from the seed of the random function.

To conclude, the tests have shown the congruency between inputs and output. The correlation between the number of Axioms and number of referenced instances, which we assumed at the beginning of this section, has been confirmed. This is the essential prerequisite to study the performances of the tool in the next section.

4.2 Performance

In the previous section we checked the output of the tool. Now the document presents the analyzis of the performances. The goal of this section is to understand the critical points of the output's creation in order to make optimizations and accelerate the response of the tool.

Theoretically, we expect to see from the tests of this section that the critical point of the tool is the creation of the Concepts array. In fact, this array can have millions of cells and it is stored in memory.

The benchmarking tests were conducted on a machine with the following configuration: Windows XP Workstation with Intel Core2 Duo T7300 2.0GHz Processor and 2 GB of physical RAM DDR2.

In the Test were used the following ontologies:

Ontology 1 (O1): 2 millions Concepts, 1 million Properties, 10 millions Individuals, 111000 Concept Axioms, 55000 Property Axioms, 5 millions Class Assertions and 0.5 million Property Assertions.

Ontology 2 (O2): the same characteristics as O1 but with 1 million Concepts.

Ontology 3 (O3): the same characteristics as O1 but with 0.5 million Concepts.

We chose these ontologies because we want to make stressful tests for the tool. The goal is to check if the tool is able to respect the requirement of creating ontologies with millions of entities and to analyze the time that OntoCreator required to do it.

Table 9: Performance tests

	O1	O2	O3
1 GB (Xmx1024M)	0:09:25	0:08:59	0:08:30
512 MB (Xmx512M)	NO	0:11:12	0:09:55

In Table 9 there are the average times for the creation of the ontologies by command line. The option "Xmx" specifies the maximum size, in bytes, of the memory allocation pool. It is important to note that ontology 1 requires more than 512 MB of physical RAM.

There is some important information about CPU occupation and memory allocation which is explained in the following figure obtained through Profiler 5.5[33] of the NetBeans IDE.

It is important to emphasize that java Profile requires resources too, so the tests are not completely equal to the situation when the tool is started from command line.

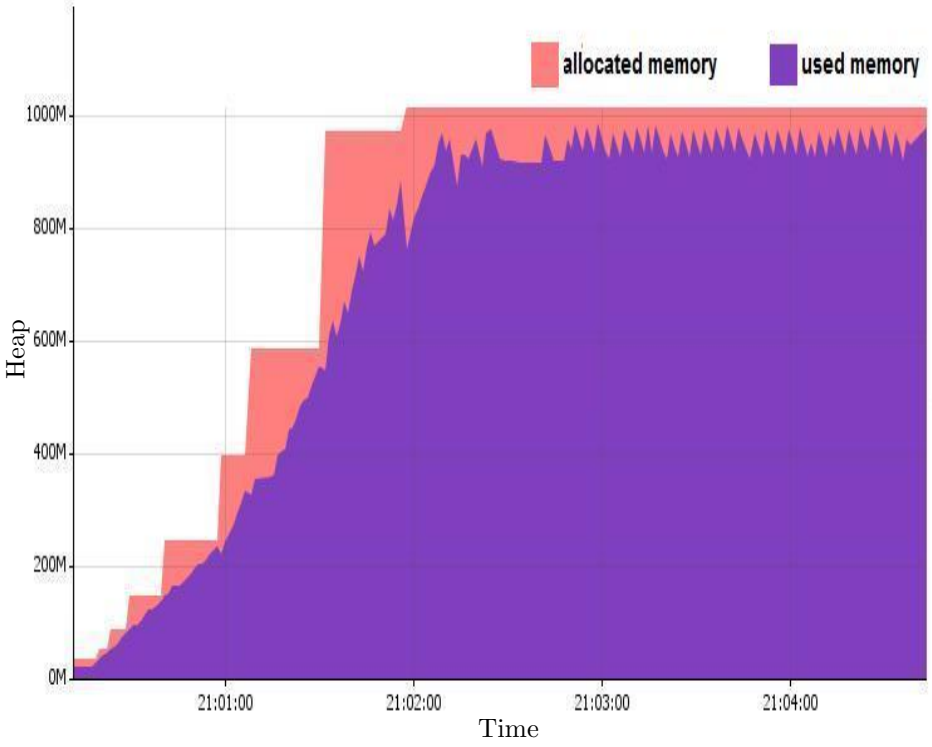

Figure 34: TEST, memory allocation I

Figure 34 shows the size of allocated memory and the size of the used memory for the creation of Ontology 1. In this case the tool used in runtime 1 GB of heap (JVM option "-Xmx1024m"). In figure, we can also appreciate that the maximum size of allocated memory is around 1000 MB, so the tool uses almost all the heap to create an ontology of two million Concepts.

[33] http://profiler.netbeans.org/

Figure 35: TEST, memory allocation II

In Figure 35 there is the memory allocation for the creation of Ontology 2. As we can see, the maximum size of allocated memory is around 700MB and it is around 400MB for the used memory. So, we expect that it is also possible to create Ontology 2 with only 512MB of dedicated RAM. In fact, the tests confirm it.

At the target point (marked with a circle) we can see that the process of ontology creation requires more quantity heap even if it does not have a real need. This is a factor that can lead to greater speed because the garbage collector can works less.

In Figure 36 it is possible to see how the garbage collector works during the creation of the Ontology 2.

The left vertical axis represents the number of the created object. The right vertical axis represents the percentage of CPU time dedicated to garbage operations.

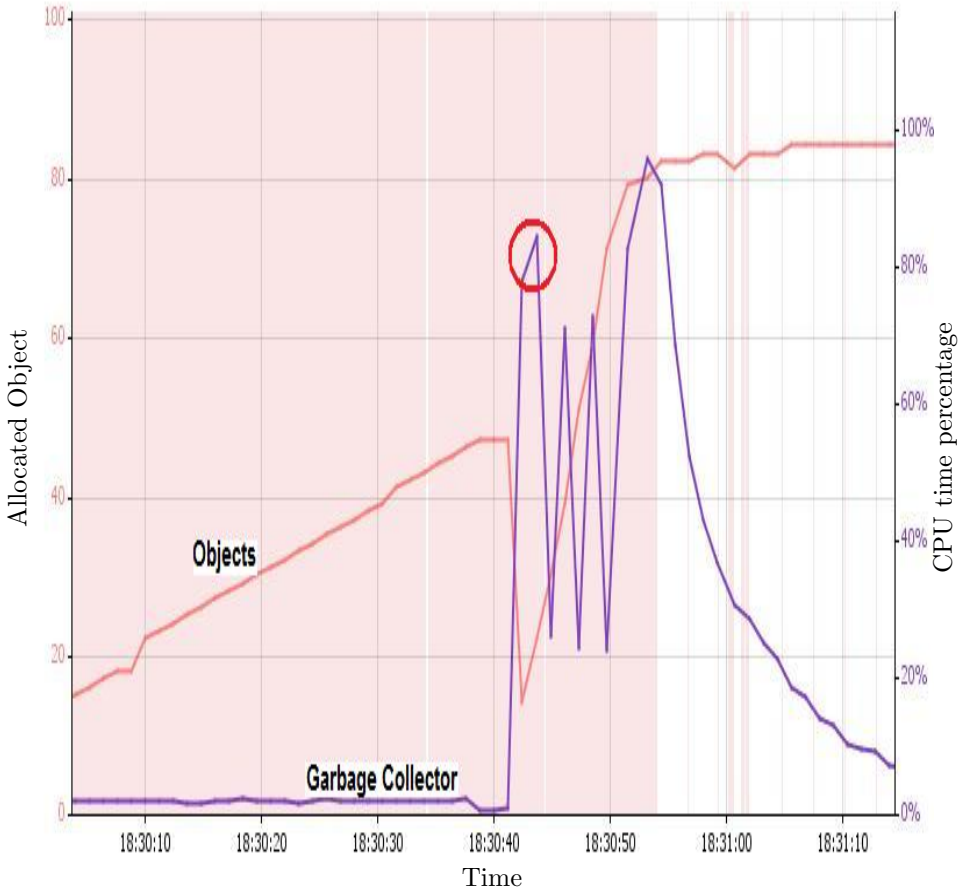

Figure 36: TEST, garbage collector

We can see here that the creation of Concepts array starts at 18:30:42. In fact, at this moment the garbage collector has a critical point (marked with a circle) because it must delete all the references to Properties and Individuals object. This is the reason because the line of active objects has a significant drop down at this moment. This is necessary so as not to exhaust the heap.

The garbage collector works hard during the creation of the array (about from 18:30:42 to 18:30:55). After this, the tool uses the Concept array to create the Axioms without storing them in memory. This is why the line of active objects remains about constant.

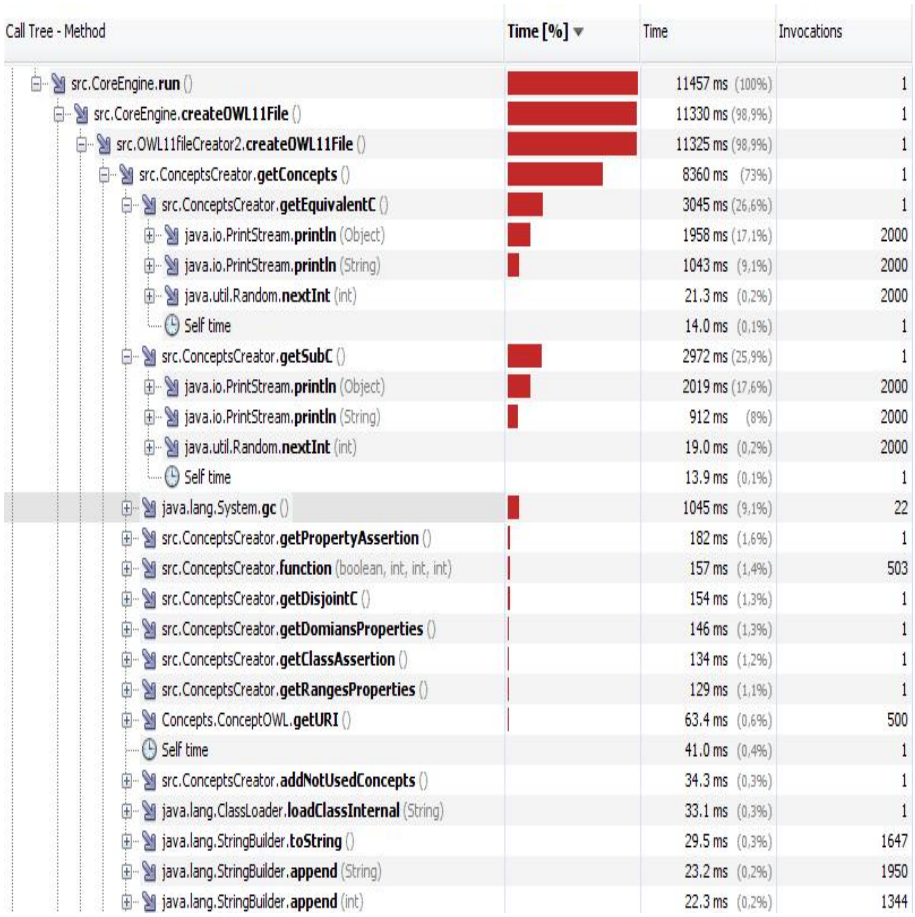

Figure 37: TEST, CPU occupation

Figure 37 shows the CPU time of all methods of the OntoCreator process to create the Ontology 1. This snapshot was taken just after having created the Sub Classes and Equivalent Classes. Here, we can see that the critical part of the OWL creation of Ontology 1 is, as we expected, the construction of the Concepts, that at this moment is already 73% of total time of the process.

As we can see, the most onerous parts after the array's creation are the *PrintStream* and the *Garbage Collector*. The first one requires more than 50%, i.e. it is 17,1% plus 9,1% for Equivalent Classes and 17,6% plus 8% for Sub Classes. The second one requires 9.1%.

Consequences

As a result of these tests we can determine that the Concepts array creation, the writing of the output file and the operations of garbage colletor are the three critical parts of the tool that we must try to improve.

We have seen that the tool can create ontology of two million Concepts in ten minutes. We can define this result as satisfactory, but we must do some tests to check if it is possible to improve it.

In the snapshot of the process time, we have seen that the method PrintStream.pritnln requires more than 50% of total time. Concerning this, we can do some tests to assess the different performances of the following methods:

1. print a String with FileOutputStream;
2. write bytes with FileOutputStream;
3. write a String with OutputStreamWriter.

The times measured with ontology of 300 thousand Concepts are:

1. 9 minutes and 14 seconds;
2. 7 minutes and 41 seconds;
3. 55 seconds.

We can see that it is about ten times faster to write a String with OutputStreamWriter than to print a String with FileOutputStream.

Consequently, the first optimization is to use an OutputStreamWriter instead of a FileOutputStream.

Concerning the memory problem, we can now check different solutions to achieve one with a lower employment of memory.

Two realizations are possible for the allocation of the unused Concepts. One solution uses the array of Boolean to store the information about them.

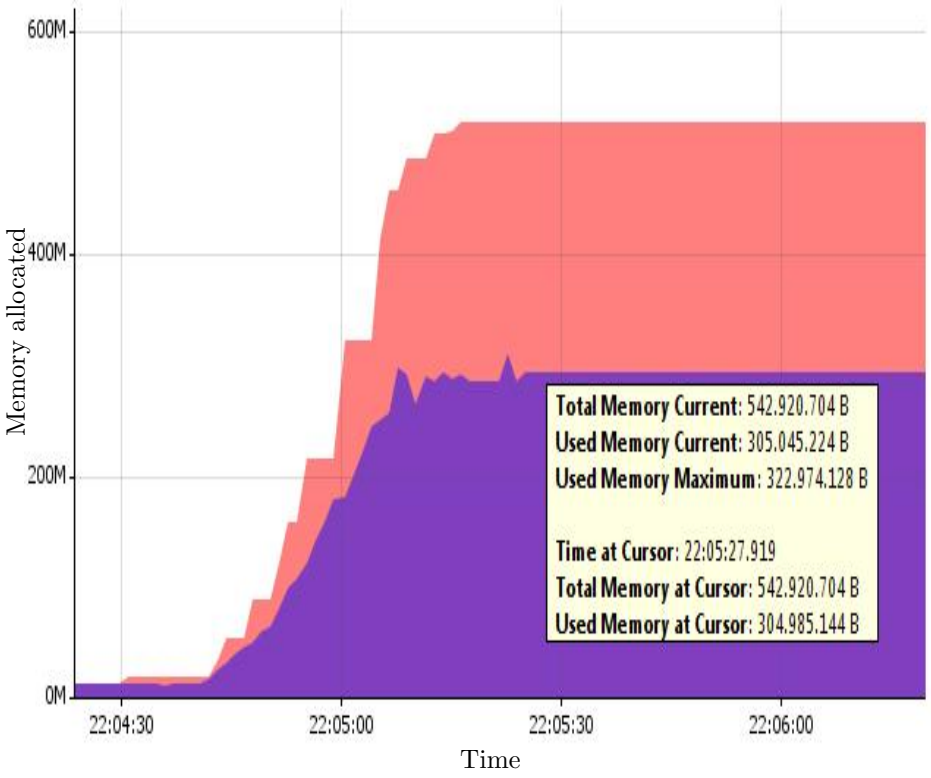

Figure 38: TEST, memory allocation III

In Figure 38 there is the memory allocation for the creation of O2 through the first solution.

A different implementation is to store this information directly into the array of Concepts.

Figure 39 shows the memory allocation of the creation of O2 with this second solution.

As we can see, the two solutions have almost the same values of used memory. The maximum value of the second deployment is minor with respect to the first solution. For this reason it was decided to implement the tool with this solution.

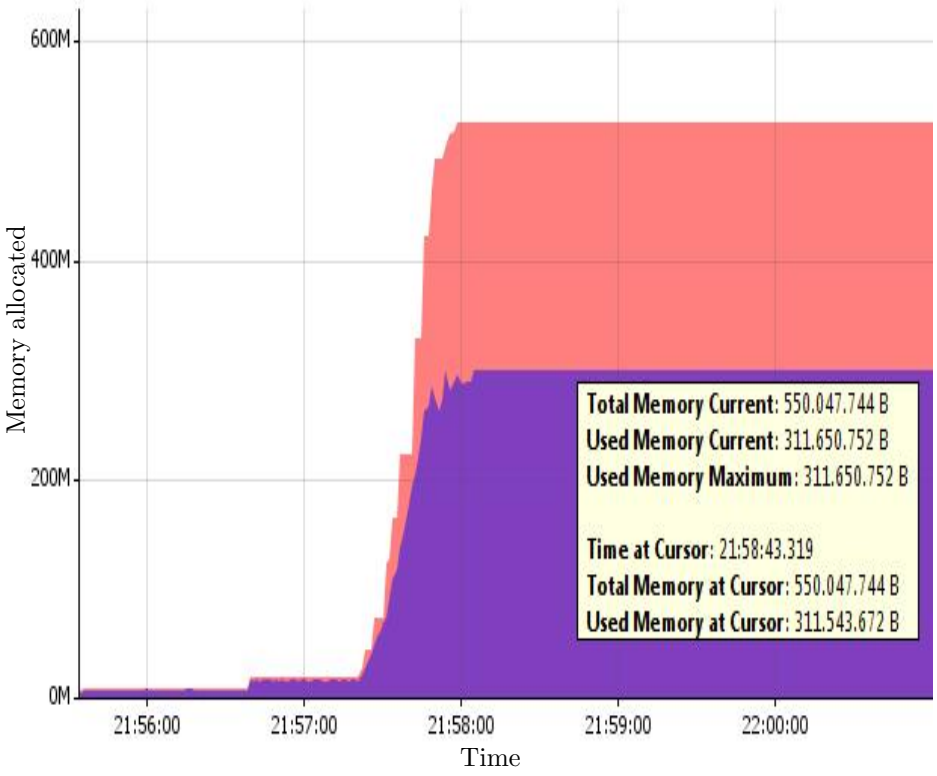

Figure 39: TEST, memory allocation IV

Table 10 shows the results of the texts made on this new implementation.

Table 10: Performance tests

	O1	O2	O3
1 GB (Xmx1024M)	0:01:34	0:0:59	0:00:48
512 MB (Xmx512M)	0:01:37	0:01:00	0:00:49

We can see in Table 10 that now the tool takes only 1 minute and 34 seconds to create Ontology 1. This is about seven times faster compared with 9 minutes and 24 seconds for the first text.

Another important improvement is that now it is also possible to create Ontology 1 with only 512 MB of heap.

The topic of the following snapshots are the memory allocation and the garbage collector's work to create an ontology of 2.5 million Concepts. The aim is to see improvements of the tool for creating an ontology which has more instances of ontology 1.

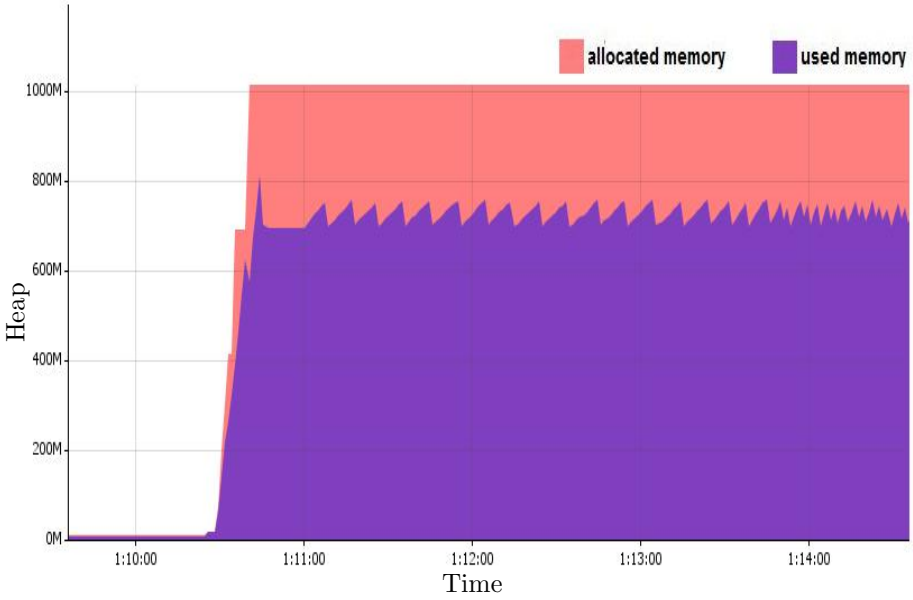

Figure 40: TEST, memory allocation V

In Figure 40 there is the memory allocation for the creation of the ontology. As we can see, now the tool uses less memory in respect to the used memory in the first test.

Concerning the time that the tool took to create the ontology, we must consider that it is more that five minutes because the java Profiler needs time and memory.

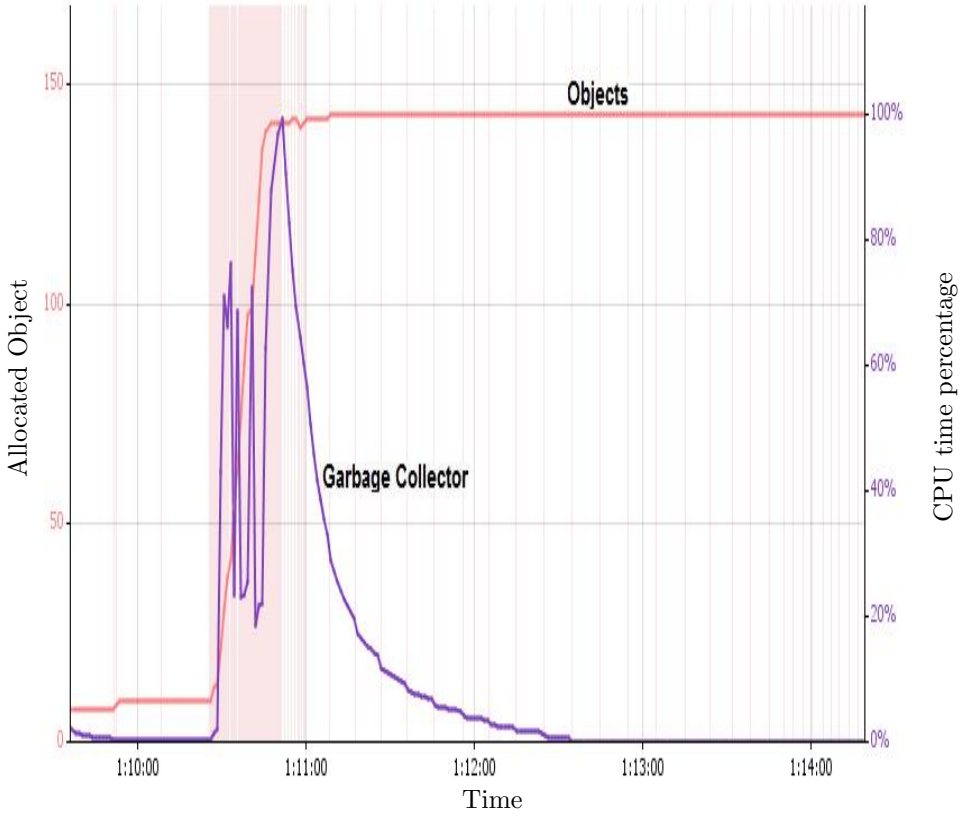

Figure 41: TEST, garbage collector II

Figure 41 shows that the garbage collector works less now with respect to the first test. In particular the critical point was around 90% in the first test, now it is around 70% of CPU time.

Summing up, we can consider these optimizations satisfactory because we have a great improvement of performance with respect to the first implementation and because we can create ontologies with millions of instances in less than a minute.

4.3 Consistency Tests with Racer

In this section we are going to analyze the consistency of ontologies produced with OntoCreator to understand the impact of the inputs on the consistency.

Given a syntactically correct OWL ontology, semantic defects can be detected by a reasoner that checks the knowledge base.

As we have already explained in Section 2.3, there are currently many reasoners, but almost none covers all the OWL1.1 specifications. We are going to use here two different reasoners to exploit the different features that they offer.

In this section we use *RacerPro*, version 1.9.2. The disadvantage of RacerPro was that when we made our tests the version of the file JRacer that it is necessary to communicate with RacerPro did not have full support for all types of Concepts. For this reason, we made tests over ontologies with only Atomic Concepts and without the unused Concepts (which can be not only Atomic Concepts).

For the tests of consistency we start to study small ontologies with thousands of entities, and afterwards check huge ontologies. This approach is necessary because it is very heavy for a reasoner to check the consistency, in fact it requires a lot of resources and time.

We want check in this section the dependency of ontologies' consistency from the inputs. In particular, we expect to obtain that the consistency depends mainly on the presence of Axioms such as Disjoint Classes, Disjoint Properties, Different Individuals because they have, respect the other Axioms, more relations with the other Axioms which can introduce inconsistency.

As a starting point, we can establish that the simplest ontology has only Concepts, Properties and Individuals, without Axioms.

It is obvious that these kinds of ontologies are always consistent because they do not presuppose possible contradictions. In fact, if we do not add the unused Concepts, the ontology can only have declaration of Individuals and Properties.

The ontology is still consistent when we insert Same Individuals, Domain Axioms, Range Axioms, Properties Assertions and Class Assertions (always of Atomic Concepts and Properties).

All tests made with these characteristics found consistent ontologies, as expected.

Individuals

The next step is to insert Individuals instantiations in the ontology to analyze the impact that they have on consistency.

Concerning Individuals, an ontology is inconsistent if it defines two Individuals as different when they are equivalent. On this subject, it is important to remember that the tool does not allow an Individual to be different to itself. Therefore, the tool can only insert in the ontology that an Individual is different to another Individual.

Table 11: Consistency test on Same Individuals and Different Individuals

	0	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
1000	100	100	100	100	100	100	100	100	100	100	100
2000	100	100	90	85	85	80	80	75	70	65	50
3000	100	95	85	80	75	70	65	60	55	50	25
4000	100	90	80	75	70	60	55	50	45	40	0
5000	100	80	60	55	35	25	20	15	10	5	0
6000	100	25	0	0	0	0	0	0	0	0	0
7000	100	0	0	0	0	0	0	0	0	0	0
8000	100	0	0	0	0	0	0	0	0	0	0
9000	100	0	0	0	0	0	0	0	0	0	0
10000	100	0	0	0	0	0	0	0	0	0	0

In Table 11, we have axis the number of Same Individuals in the horizontal and the number of Different Individuals in the vertical axis. Inside the table we have the percentage of consistent ontologies with these characteristics.

The tests were performed on ontology with 10.000 Individuals. As expected, if the numbers of Same Individuals or the numbers of Different Individuals, is equal to zero, the ontology is certainly consistent.

Another characteristic that we expected is that, if the number of Same or Different Individuals increases, the percentage of consistent ontologies decreases.

Figure 42 is the three-dimensional representation of Table 11. As we can see, the table is not diagonally symmetrical. This is because there are more checks on the Different Individuals with respect to the Same Individuals. In fact, an ontology cannot have an Individual that is different from itself.

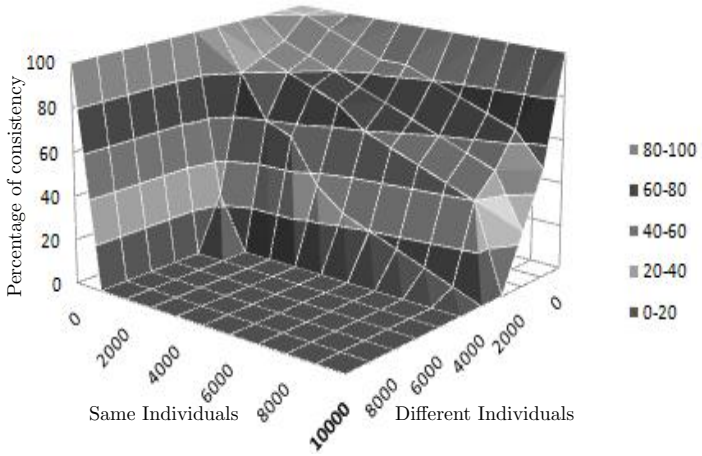

Figure 42: Consistency, Same Individuals-Different Individuals

Another possibility for inconsistency involving Individuals is if there are simultaneously Same Individuals, Disjoint Classes and Class Assertion in the ontology.

```
<SameIndividuals>
    <Individual URI="&ns;I0"/>
    <Individual URI="&ns;I1"/>
</SameIndividuals>
<DisjointClasses>
    <OWLClass URI="&ns;C0"/>
    <OWLClass URI="&ns;C1"/>
</DisjointClasses>
<ClassAssertion>
    <Individual URI="&ns;I0"/>
    <OWLClass URI="&ns;C0"/>
</ClassAssertion>
<ClassAssertion>
    <Individual URI="&ns;I1"/>
    <OWLClass URI="&ns;C1"/>
</ClassAssertion>
```

With this code, we expect the ontology to be inconsistent; in fact, the tests confirm this. The same thing could occur with a Class, its ComplementOf and an Individual that belongs to both the Concepts.

Table 12: Consistency test on Same Individuals and Disjoint Classes

	0	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
0	100	-	-	-	-	-	-	-	-	-	-
1000	100	95	95	95	90	80	85	90	80	80	85
2000	100	90	100	95	100	95	90	95	90	85	90
3000	100	95	100	100	100	90	95	100	90	85	95
4000	100	90	100	95	95	95	100	95	100	90	95
5000	100	100	100	100	100	95	95	100	95	95	100
6000	100	95	100	100	100	100	95	95	90	100	100
7000	100	100	100	100	100	100	100	85	95	90	100
8000	100	100	100	95	100	100	100	100	100	95	100
9000	100	95	100	100	100	100	100	100	100	100	95
10000	100	95	100	100	100	100	95	100	95	100	100

Table 12 shows the results of the tests made on ontologies containing Same Individuals, Class Assertions and Disjoint Classes.

In the horizontal axis of Table 12 there are the number of Same Individuals and in the vertical axis the number of Disjoint Classes. The number of Disjoint Classes is equal to the number of Class Assertions and the total number of Classes. The total number of Individuals is equal to the number of Same Individuals multiplied by ten.

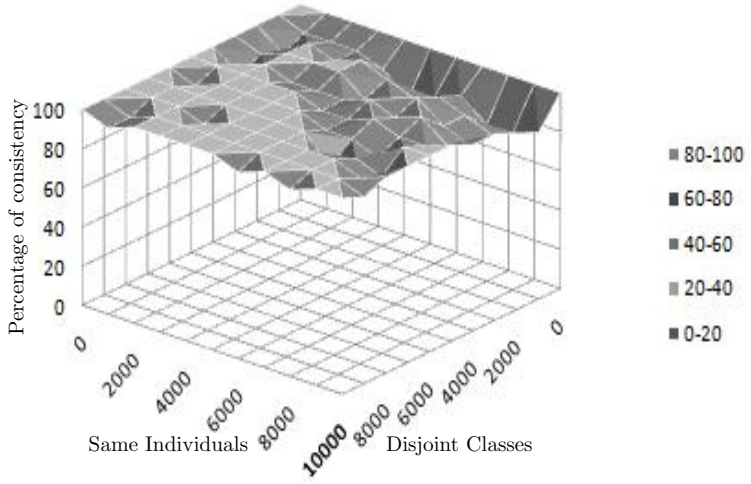

Figure 43: Consistency, Same Individuals-Disjoint Classes

Figure 43 is the three-dimensional representation of Table 12.

As expected, if the number of Classes is zero, the percentage of consistent ontologies is 100%. If the number of Individuals is zero, the ontology cannot have a Class Assertion. The results show that the percentage of consistent ontology is always over 80%.

Properties

Concerning Properties, a possible cause of inconsistency can be:

```
<EquivalentObjectProperties>
<ObjectProperty URI="&ns;P0"/>
<ObjectProperty URI="&ns;P1"/>
</EquivalentObjectProperties>

<DisjointObjectProperties>
<ObjectProperty URI="&ns;P0"/>
<ObjectProperty URI="&ns;P1"/>
</DisjointObjectProperties>

<ObjectPropertyAssertion>
<Individual URI="&ns;I0"/>
<Individual URI="&ns;I1"/>
<ObjectProperty URI="&ns;P0"/>
</ObjectPropertyAssertion>

<ObjectPropertyAssertion>
<Individual URI="&ns;I0"/>
<Individual URI="&ns;I1"/>
<ObjectProperty URI="&ns;P1"/>
</ObjectPropertyAssertion>
```

Unfortunately, the results of tests show that the ontology is consistent. In fact, all the tests on inconsistencies caused by Properties gave the same answer. The reason is that RacerPro does not check the possible inconsistency introduced by the relations between the Properties.

Classes

The Classes are the most relevant cause of inconsistency. In fact, there are many types of Classes that may be involved in contradictions.

The problem is that at the moment we can only tests Axioms of Atomic Concepts, so, the probability of inconsistency is less with respect to ontologies that include all types of classes.

A possible cause of inconsistency is:

```
<SubClassOf>
    <OWLClass URI="&ns;C0"/>
    <OWLClass URI="&ns;C1"/>
</SubClassOf>

<DisjointClasses>
    <OWLClass URI="&ns;C0"/>
    <OWLClass URI="&ns;C1"/>
</DisjointClasses>
```

In this case, we expected the reasoner to show that the two Classes are incoherent and the ontology is consistent. However, the tests show that the ontology is inconsistent.

Table 13: Consistency test on Sub Classes and Disjoint Classes

	0	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
0	100	100	100	100	100	100	100	100	100	100	100
1000	100	95	85	80	80	80	80	80	80	80	80
2000	100	90	70	60	60	60	60	60	60	60	60
3000	100	85	55	55	45	45	40	40	40	40	30
4000	100	80	50	50	40	40	40	40	40	35	25
5000	100	75	45	25	20	15	10	10	10	10	10
6000	100	50	10	10	0	0	0	0	0	0	0
7000	100	45	0	0	0	0	0	0	0	0	0
8000	100	20	0	0	0	0	0	0	0	0	0
9000	100	0	0	0	0	0	0	0	0	0	0
10000	100	0	0	0	0	0	0	0	0	0	0

In the horizontal axis of Table 13 there are the number of Sub Classes; in the vertical axis there are the number of Disjoint Classes. Inside the table there is the percentage of ontology consistency. The tests were made on ontology with 10.000 Concepts.

As expected, if the number of Sub Classes, or the number of Disjoint Classes, is equal to zero, the percentage of consistency is always 100%. Another feature that we expected is that, on increasing the number of Axioms, the percentage of consistency decreases.

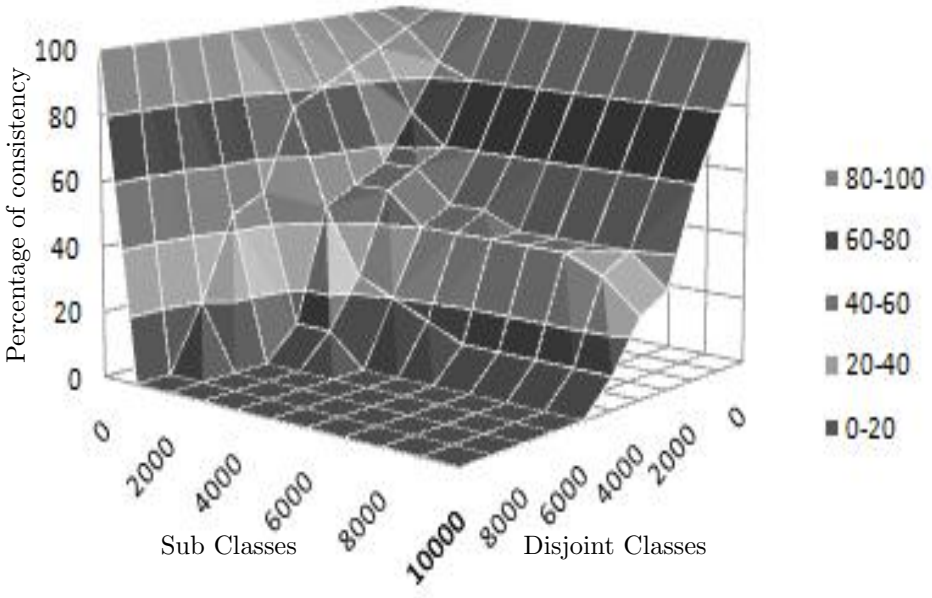

Figure 44: Consistency, sub Classes-disjoint Classes

Figure 44 is the three-dimensional representation of Table 13. Again, the table is not diagonally symmetric because Disjoint Classes introduce more contradictions with respect to Sub Classes.

Individuals, Properties and Concepts

Finally, we made tests on ontologies with Individuals, Properties and Concepts together. We wanted to check the relations among the different entities. The goal is to check how the Axioms interact between themselves and understand which percentages of the different entities guarantee consistent ontologies are obtained.

We expected the percentage of consistency for this ontology not to be too big, since it depends on the percentage between Same Individuals and Different Individuals (35%) and between Sub Classes and Disjoint Classes (55%).

The tests were performed on ontologies made by a metadata file with the following inputs:
Total number of Atomic Concepts: 1.000;
Number of sub Classes: 300;
Number of disjoint Classes: 300;
Total number of Atomic Properties: 1.000;
Number of functional Properties: 100;
Number of reflexive Properties: 100;
Number of symmetric Properties: 100;
Number of transitive Properties: 100;
Number of inverse functional Properties: 100;
Number of irreflexive Properties: 100;
Number of antisymmetric Properties: 100;
Number of sub Properties: 100;
Number of disjoint Properties: 100;
Number of inverse Properties: 100;
Number of domain Properties: 100;
Number of range Properties: 100;
Total number of individual: 1.000;
Number of same Individuals: 400;
Number of different Individuals: 400;
Number of class assertions: 300;
Number of property assertion: 350;

The tests showed that the percentage is 100%, but If any of these critical inputs is increased by 50 (by 5% of total number of entities) virtually all generated ontologies will be inconsistent. This means that the Axioms are strongly correlated.

If we test the same ontology with the inputs multiplied by ten, the ontology is still consistent. This means that, as expected, the consistency is a characteristic that depends on the percentages of relations and not on specific numbers of inputs.

The tests have shown that, to have a good percentage (greater than 50%) of consistency, it is necessary for the number of Axioms not to be greater than 30% of the total number of entities. This can be considered satisfactory because it means that in ontologies with a million Concepts there may be three hundred thousand of each kind of Axiom.

4.4 Consistency tests with Pellet

After having seen the limits of Racer, we can now analyze OntoCreator with Pellet. Pellet is another reasoner for OWL1.1. The current stable version of Pellet is 1.5.1. Pellet has still problem to support all possible Axioms of Concepts at the moment. The advantage of Pellet, compared to Racer, is that it controls the relationship between the Properties.

The first step was to check the same features tested with Racer. We expected, for Pellet, the same results achieved with Racer. In fact, the tests on Individuals and on Classes gave us the identical results.

An objective characteristic, which was observed during the tests, is the greater memory occupation and slower speed of Pellet compared to Racer. In particular, with ontologies of one thousand Classes or one thousand Individuals the tests responded with StackOverflowError. For this reason we added, for the run-time of the JVM, the option -Xss2048k, which increases the default size of the stack. Even with this change, for tests with ontologies of ten thousand Classes, Pellet consumes all the 1024MB of heap.

Unlike Racer, Pellet does not warn that the ontology is inconsistent, but it launches an Inconsistent Ontology Exception.

Properties

Table 14: Consistency test on Disjoint Properties and Property Assertions

	0	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
0	100	100	100	100	100	100	100	100	100	100	100
1000	100	100	95	90	85	80	75	70	55	55	50
2000	100	95	90	80	65	65	60	30	25	20	15
3000	100	90	80	65	55	50	45	30	20	15	10
4000	100	85	70	55	50	50	35	25	15	10	10
5000	100	80	65	50	50	45	30	15	10	10	10
6000	100	75	60	50	45	30	20	10	5	5	5
7000	100	70	50	45	30	30	15	10	5	5	5
8000	100	65	55	35	30	20	10	5	5	5	5
9000	100	55	40	30	25	20	10	5	5	5	0
10000	100	50	35	30	20	15	10	5	5	0	0

In the horizontal axis of Table 14 there are the numbers of Disjoint Properties; in the vertical axis there are the numbers Property Assertions. Inside the table there is the percentage of ontology consistency.

As expected, if the number of Disjoint Properties, or the number of Property Assertions, is equal to zero, the percentage of consistency is always 100% (always consistent).

Another feature that we expected is that, when the number of Axioms increases, the percentage of consistency decreases.

Figure 45 is the three-dimensional representation of Table 14.

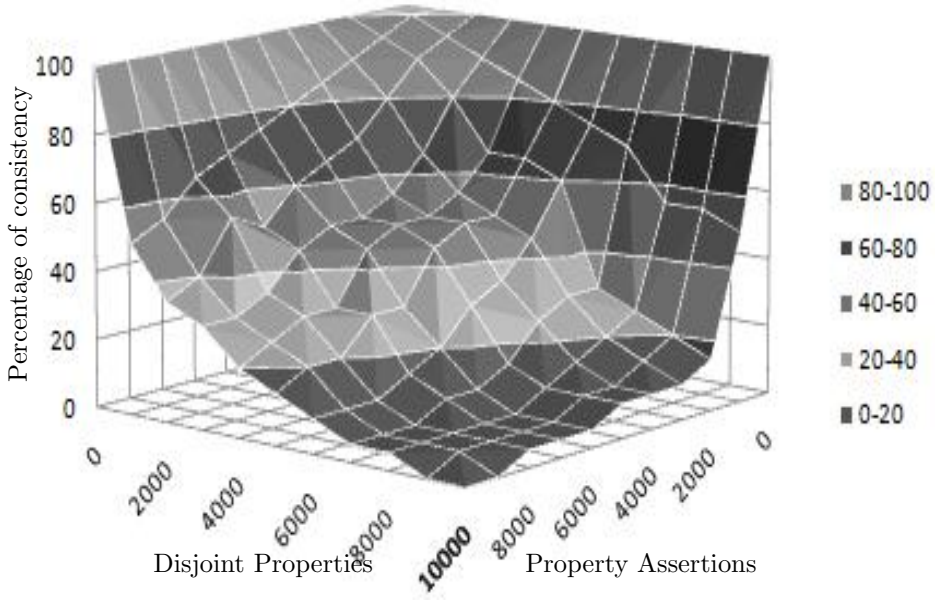

Figure 45: Consistency, Disjoint Classes-Property Assertions

Concerning Properties, other causes that may introduce inconsistencies are:

- If a property is symmetric and it is anti-symmetric, if they are not empty Properties the ontology is inconsistent;
- If a property is defined as functional (or inverse-functional) and it has an individual connected with more than one individual;
- If a property is irreflexive and it has an individual correlated to himself;

Concepts, Properties and Individuals

The last step is to test ontologies with Concepts, Properties and Individuals together. Again, the goal of these tests is to check which percentage of Axioms can be certain to have consistent ontologies. It is important remember that the consistency also depends on the seed of the random function. Therefore, it is impossible to test the application with all possible seeds because it would take a very long time.

The results of these tests showed that all the various types of Axioms should be around 3% of the total number of the respective entities (Concepts, Properties and Individuals) to ensure a consistent ontology. This result is so much lower with respect to the 30% recorded by Racer. However, we must remember that with Racer we could not check the relationship introduced by the Properties.

If we test ontologies without Functional, Antisymmetric and Irreflexive Properties, the results of the tests show that all the various types of Axioms should be 10% of the total number of the category's entities to have a consistent ontology. We can consider this result as satisfactory because it means that, for example, with ontologies of one million Concepts there are one hundred thousand Disjoint Concepts (the same number for other Axioms).

It is possible to have consistent ontologies with a higher number of Axioms. However, the percentage of consistency decreases quickly mainly because of Disjoint Classes and Properties.

4.5 Summary

After these tests, we can now compare the solutions obtained with RacerPro and Pellet.

The main difference between them is that with Pellet we were able to test if Properties introduce inconsistency. For the other tests, we have obtained a congruence of results with the two different reasoner.

Summing up, we have seen that the inputs are strongly connected and the consistency of ontologies depends mainly on the presence of Axioms such as Disjoint Classes, Disjoint Properties, Different Individuals because they have several relations with other kinds of Axioms which can introduce inconsistency.

5 Conclusions and FurtherWork

In this thesis, a tool able to automatically create synthetic ontologies was discussed, justified and formally produced.

In this document, we have seen that there is a low quantity of ontologies available at the moment, especially for huge ontologies. The few tools that can produce ontologies have several problems; for example, they produce data no interconnected or they do not use all possible types of entities.

Furthermore, this paper presented an overview of several already existing reasoners. They have been surveyed and evaluated.

Therefore, it is necessary to have a tool able to generate ontologies from given characteristics. This is, as we saw reviewing the matter in the first Chapters, something of new in the area of semantic application and it is a need not covered.

These are some motivations to develop OntoCreator. In addition, through this tool it is possible to create huge ontologies to be used for benchmarking. This is one of the main results of this thesis.

OntoCreator was created on the specific request of the University of Ulm. However, as explained, this work is useful in general to implement reasoners and benchmarks. Therefore, this tool can contribute to the development of the semantic applications area.

Another goal of this tool is to easily share huge ontologies, without being required to send a file with the size of gigabytes. So, we have seen that a solution can be to share only a file of metadata where are stored all the information of the creation's process. We have also introduced Java Web Start to avoid problems of versioning and uploading. In this way, we have the possibility to rebuild the ontology on another computer or in another moment because the tool can always produce the same output file from the same inputs.

More, we discussed in this paper the possibility of using OWL API and its problems. We have seen that it is a memory limit to the creation of huge ontologies and then we have decided to use generic java classes that take care of the output syntax. In this way, it will be possible to easily extend the tool adding other classes in the future. In fact, other classes can represent the semantic with another syntax.

In addition, during the analysis of tools, we described how to optimize the performance of tools and, we especially studied the structure of ontologies and the relationship between their entities.

During the evaluation of the tool, we saw that various kinds of Axioms are strongly correlated. Indeed, we can only be sure of obtaining consistent ontologies with a percentage of 3% for the different Axioms. We have also seen that the number of referenced instances are directly proportional to the number of Axioms and that, to reference all the instances of entities, is necessary a number of Axioms greater than the number of instances. This is because the Axioms can use the same entities several times.

The main objective of possible future work is to expand the first version of the tool with a different output file, such as OWL1.0.

Other future works could be to provide OntoCreator as a plug-in for Protégé or to extend the tool in another language.

Concerning the future of semantic applications, the most important requirement is to have reasoners and benchmarks that fully cover the OWL specifications.

Acknowledgments

It is a great pleasure to acknowledge the sincere and helpful advice of Prof. Dr. Friedrich von Henke and Timo Weithner, Ulm University, who helped accomplish this thesis and finally assessed it as the chair of the reviewing committee in Bologna University. The author also thanks Prof. Paola Mello, member of the Italian committee, for her help in carrying out this thesis. A special thanks to my family, for the continuous encouragement they gave me. A special thanks to Charlottina, who accompanied me for a year during the realization of the thesis. I also want to thank all my friends in Bologna, for all the times they were near me and all the friends that i found here in Ulm for all the great time we had together.

Appendixes

A Medadata File 1

RangeProp	10
RangeOnlyAtomcConceptsCheckBox	false
TransitivePropCheckBox	false
InversePropPerc	20
DomainProp	10
ReflexivePropCheckBox	false
SomeValueConceptsCheckBox	false
ExactQCRPerc	60
AtomicPropPerc	80
MaxUnQCROnlyAtomcPropCheckBox	false
DisjointPropCheckBox	false
IntersectionOfCPerc	5
IndividualsNum	100
CardUnqualifCPerc	5
ExactQCROnlyAtomcPropCheckBox	false
DisjointClasses	10
MinRangeEQC	2
MaxUnionMembers	5
PropertyAssertionsCheckBox	false
MinRangeEUnQC	2
MinRangeMinQC	0
ReflexiveProp	10
MaxRangeEUnQC	5
MaxInterMembers	5
MaxQCROnlyAtomcPropCheckBox	false
MaxRangeMinQC	3
MinOneOfMembers	2
CardQualifCPerc	5
ExactQCROnlyAtomcConceptsCheckBox	false
MinQCROnlyAtomcPropCheckBox	false
MaxRangeMaxUnQC	5
MaxQCROnlyAtomcConceptsCheckBox	false
SubPropCheckBox	false

EquivalentClasses	10
AntiSymmetricPropCheckBox	false
MaxRangeMinUnQC	3
MinUnQCROnlyAtomcConceptsCheckBox	false
IrreflexivePropCheckBox	false
UnionOfConceptsCheckBox	false
EquivalentProp	10
RangeOnlyAtomcPropCheckBox	false
ExistsSelfCPerc	5
DisjointProp	10
SubClasses	10
MaxUnQCROnlyAtomcConceptsCheckBox	false
OneOfCPerc	5
PropertyAssertions	10
DomainOnlyAtomcConceptsCheckBox	false
ExactUnQCROnlyAtomcPropCheckBox	false
IntersectionOfCheckBox	false
InversePropCheckBox	false
TotalNumC	100
TotalNumPro	100
MaxQCRPerc	20
SomeValueCPerc	5
IrreflexiveProp	10
TransitiveProp	10
FunctionalPropCheckBox	false
SymmetricPropCheckBox	false
MaxRangeEQC	5
ExistsSelfCheckBox	false
MinUnionMembers	2
DomainOnlyAtomcPropCheckBox	false
ClassAssertionsCheckBox	false
SymmetricProp	10
MinQCRPerc	20
SomeValuePropCheckBox	false
UnionOfCPerc	5

MinUnQCRPerc	20
AllValueConceptsCheckBox	false
MinRangeMaxQC	2
ExactUnQCRPerc	60
HasValueCheckBox	false
SameIndiv	10
MinRangeMaxUnQC	2
MaxRangeMaxQC	5
FunctionalProp	10
SeedRandomFunction	922337203
DifferentIndiv	10
EquivalentPropCheckBox	false
MinInterMembers	2
AllValueCPerc	5
ClassAssertions	10
MaxUnQCRPerc	20
MinUnQCROnlyAtomcPropCheckBox	false
AllValuePropCheckBox	false
InverseProp	10
HasValueCPerc	5
ExactUnQCROnlyAtomcConceptsCheckBox	false
MinQCROnlyAtomcConceptsCheckBox	false
InverseFunctionalPropCheckBox	false
InverseFunctionalProp	10
MinRangeMinUnQC	0
SubProp	10
AtomicCPerc	50
MaxOneOfMembers	5
ComplementOfCPerc	5
ComplementOfCCheckBox	false
AntiSymmetricProp	10

B Ontology 1

```
<?xml version="1.0"?>
<!DOCTYPE Ontology [
<!ENTITY owl "http://www.w3.org/2002/07/owl#" >
<!ENTITY owl11 "http://www.w3.org/2006/12/owl11#" >
<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
<!ENTITY owl11xml "http://www.w3.org/2006/12/owl11-xml#" >
<!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
<!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
<!ENTITY ns "http://www.co-ode.org/ontologies/onto.xml#" >
]>
<Ontology xmlns="http://www.w3.org/2006/12/owl11-xml#"
xml:base="http://www.w3.org/2006/12/owl11-xml#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:owl11="http://www.w3.org/2006/12/owl11#"
xmlns:ns="http://www.co-ode.org/ontologies/onto.xml#"
xmlns:owl11xml="http://www.w3.org/2006/12/owl11-xml#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:owl="http://www.w3.org/2002/07/owl#"
URI="http://www.co-ode.org/ontologies/onto.xml">

<Declaration>

<Individual URI="&ns;I0"/>

<Individual URI="&ns;I1"/>

</Declaration>

<SameIndividuals>
<Individual URI="&ns;I0"/>
<Individual URI="&ns;I4"/>
</SameIndividuals>

<DifferentIndividuals>
<Individual URI="&ns;I3"/>
```

```
<Individual URI="&ns;I1"/>
</DifferentIndividuals>

<Declaration>

<ObjectProperty URI="&ns;P0"/>

</Declaration>

<FunctionalObjectProperty>
<InverseObjectProperty>
<ObjectProperty URI="&ns;P1"/>
</InverseObjectProperty>
</FunctionalObjectProperty>

<ReflexiveObjectProperty>
<ObjectProperty URI="&ns;P0"/>
</ReflexiveObjectProperty>

<SymmetricObjectProperty>
<ObjectProperty URI="&ns;P0"/>
</SymmetricObjectProperty>

<TransitiveObjectProperty>
<InverseObjectProperty>
<ObjectProperty URI="&ns;P1"/>
</InverseObjectProperty>
</TransitiveObjectProperty>

<InverseFunctionalObjectProperty>
<ObjectProperty URI="&ns;P0"/>
</InverseFunctionalObjectProperty>

<IrreflexiveObjectProperty>
<ObjectProperty URI="&ns;P0"/>
</IrreflexiveObjectProperty>

<AntisymmetricObjectProperty>
```

```
<InverseObjectProperty>
<ObjectProperty URI="&ns;P1"/>
</InverseObjectProperty>
</AntisymmetricObjectProperty>

<InverseObjectProperties>
<InverseObjectProperty>
<ObjectProperty URI="&ns;P1"/>
</InverseObjectProperty>
<ObjectProperty URI="&ns;P0"/>
</InverseObjectProperties>

<SubObjectPropertyOf>
<ObjectProperty URI="&ns;P0"/>
<InverseObjectProperty>
<ObjectProperty URI="&ns;P1"/>
</InverseObjectProperty>
</SubObjectPropertyOf>

<EquivalentObjectProperties>
<ObjectProperty URI="&ns;P0"/>
<InverseObjectProperty>
<ObjectProperty URI="&ns;P1"/>
</InverseObjectProperty>
</EquivalentObjectProperties>

<DisjointObjectProperties>
<InverseObjectProperty>
<ObjectProperty URI="&ns;P1"/>
</InverseObjectProperty>
<ObjectProperty URI="&ns;P0"/>
</DisjointObjectProperties>

<SubClassOf>
<ObjectHasValue>
<ObjectProperty URI="&ns;P1"/>
<Individual URI="&ns;I3"/>
</ObjectHasValue>
```

```
<ObjectHasValue>
<ObjectProperty URI="&ns;P0"/>
<Individual URI="&ns;I4"/>
</ObjectHasValue>
</SubClassOf>

<SubClassOf>
<OWLClass URI="&ns;C31"/>
<ObjectExistsSelf>
<ObjectProperty URI="&ns;P0"/>
</ObjectExistsSelf>
</SubClassOf>

<EquivalentClasses>
<ObjectHasValue>
<ObjectProperty URI="&ns;P0"/>
<Individual URI="&ns;I3"/>
</ObjectHasValue>
<ObjectAllValuesFrom>
<ObjectProperty URI="&ns;P1"/>
<OWLClass URI="&ns;C28"/>
</ObjectAllValuesFrom>
</EquivalentClasses>

<EquivalentClasses>
<ObjectExactCardinality cardinality="3">
<ObjectProperty URI="&ns;P1"/>
<OWLClass URI="&ns;C28"/>
</ObjectExactCardinality>
<OWLClass URI="&ns;C14"/>
</EquivalentClasses>

<DisjointClasses>
<ObjectUnionOf>
<OWLClass URI="&ns;C32"/>
<OWLClass URI="&ns;C12"/>
</ObjectUnionOf>
<OWLClass URI="&ns;C17"/>
```

```
</DisjointClasses>

<DisjointClasses>
<OWLClass URI="&ns;C7"/>
<ObjectExistsSelf>
<ObjectProperty URI="&ns;P0"/>
</ObjectExistsSelf>
</DisjointClasses>

<ClassAssertion>
<Individual URI="&ns;I4"/>
<OWLClass URI="&ns;C25"/>
</ClassAssertion>

<ObjectPropertyAssertion>
<Individual URI="&ns;I2"/>
<Individual URI="&ns;I0"/>
<ObjectProperty URI="&ns;P0"/>
</ObjectPropertyAssertion>

<ObjectPropertyDomain>
<ObjectProperty URI="&ns;P1"/>
<ObjectExistsSelf>
<ObjectProperty URI="&ns;P0"/>
</ObjectExistsSelf>
</ObjectPropertyDomain>

<ObjectPropertyRange>
<ObjectProperty URI="&ns;P1"/>
<OWLClass URI="&ns;C25"/>
</ObjectPropertyRange>

<SubClassOf>
<OWLClass URI="&ns;C0"/>
<OWLClass URI="&ns;Thing"/>

<SubClassOf>
<ObjectOneOf>
```

```
<Individual URI="&ns;I4"/>
<Individual URI="&ns;I2"/>
</ObjectOneOf>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectComplementOf>
<OWLClass URI="&ns;C30"/>
</ObjectComplementOf>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectComplementOf>
<ObjectUnionOf>
<OWLClass URI="&ns;C11"/>
<OWLClass URI="&ns;C6"/>
<OWLClass URI="&ns;C14"/>
</ObjectUnionOf>
</ObjectComplementOf>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectComplementOf>
<ObjectHasValue>
<ObjectProperty URI="&ns;P1"/>
<Individual URI="&ns;I2"/>
</ObjectHasValue>
</ObjectComplementOf>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectComplementOf>
<ObjectAllValuesFrom>
<ObjectProperty URI="&ns;P1"/>
```

```
<ObjectOneOf>
<Individual URI="&ns;I0"/>
<Individual URI="&ns;I3"/>
</ObjectOneOf>
</ObjectAllValuesFrom>
</ObjectComplementOf>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectIntersectionOf>
<OWLClass URI="&ns;C26"/>
<ObjectMaxCardinality cardinality="3">
<ObjectProperty URI="&ns;P0"/>
<OWLClass URI="&ns;C26"/>
</ObjectMaxCardinality>
</ObjectIntersectionOf>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectIntersectionOf>
<ObjectSomeValuesFrom>
<ObjectProperty URI="&ns;P0"/>
<OWLClass URI="&ns;C15"/>
</ObjectSomeValuesFrom>
<OWLClass URI="&ns;C38"/>
</ObjectIntersectionOf>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectIntersectionOf>
<OWLClass URI="&ns;C26"/>
<OWLClass URI="&ns;C6"/>
</ObjectIntersectionOf>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>
```

```
<SubClassOf>
<ObjectIntersectionOf>
<ObjectSomeValuesFrom>
<ObjectProperty URI="&ns;P0"/>
<ObjectSomeValuesFrom>
<ObjectProperty URI="&ns;P0"/>
<OWLClass URI="&ns;C15"/>
</ObjectSomeValuesFrom>
</ObjectSomeValuesFrom>
<ObjectIntersectionOf>
<OWLClass URI="&ns;C26"/>
<OWLClass URI="&ns;C6"/>
</ObjectIntersectionOf>
<OWLClass URI="&ns;C10"/>
</ObjectIntersectionOf>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectUnionOf>
<ObjectExactCardinality cardinality="2">
<ObjectProperty URI="&ns;P0"/>
<ObjectAllValuesFrom>
<ObjectProperty URI="&ns;P0"/>
<ObjectOneOf>
<Individual URI="&ns;I0"/>
<Individual URI="&ns;I3"/>
</ObjectOneOf>
</ObjectAllValuesFrom>
</ObjectExactCardinality>
<OWLClass URI="&ns;C29"/>
<OWLClass URI="&ns;C23"/>
<ObjectSomeValuesFrom>
<ObjectProperty URI="&ns;P1"/>
<OWLClass URI="&ns;C8"/>
</ObjectSomeValuesFrom>
</ObjectUnionOf>
```

```
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectUnionOf>
<OWLClass URI="&ns;C20"/>
<ObjectExactCardinality cardinality="3">
<ObjectProperty URI="&ns;P0"/>
</ObjectExactCardinality>
<OWLClass URI="&ns;C22"/>
<OWLClass URI="&ns;C2"/>
</ObjectUnionOf>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectUnionOf>
<OWLClass URI="&ns;C36"/>
<ObjectExactCardinality cardinality="2">
<ObjectProperty URI="&ns;P1"/>
</ObjectExactCardinality>
</ObjectUnionOf>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectSomeValuesFrom>
<ObjectProperty URI="&ns;P1"/>
<ObjectUnionOf>
<OWLClass URI="&ns;C20"/>
<ObjectExactCardinality cardinality="3">
<ObjectProperty URI="&ns;P0"/>
</ObjectExactCardinality>
<OWLClass URI="&ns;C22"/>
<OWLClass URI="&ns;C2"/>
</ObjectUnionOf>
</ObjectSomeValuesFrom>
<OWLClass URI="&ns;Thing"/>
```

```
</SubClassOf>

<SubClassOf>
<ObjectSomeValuesFrom>
<ObjectProperty URI="&ns;P0"/>
<ObjectComplementOf>
<ObjectUnionOf>
<OWLClass URI="&ns;C11"/>
<OWLClass URI="&ns;C6"/>
<OWLClass URI="&ns;C14"/>
</ObjectUnionOf>
</ObjectComplementOf>
</ObjectSomeValuesFrom>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectSomeValuesFrom>
<ObjectProperty URI="&ns;P0"/>
<OWLClass URI="&ns;C23"/>
</ObjectSomeValuesFrom>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectSomeValuesFrom>
<ObjectProperty URI="&ns;P1"/>
<OWLClass URI="&ns;C8"/>
</ObjectSomeValuesFrom>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectAllValuesFrom>
<ObjectProperty URI="&ns;P1"/>
<ObjectComplementOf>
<ObjectUnionOf>
<OWLClass URI="&ns;C11"/>
```

```
<OWLClass URI="&ns;C6"/>
<OWLClass URI="&ns;C14"/>
</ObjectUnionOf>
</ObjectComplementOf>
</ObjectAllValuesFrom>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectAllValuesFrom>
<ObjectProperty URI="&ns;P1"/>
<ObjectIntersectionOf>
<ObjectSomeValuesFrom>
<ObjectProperty URI="&ns;P0"/>
<OWLClass URI="&ns;C15"/>
</ObjectSomeValuesFrom>
<OWLClass URI="&ns;C38"/>
</ObjectIntersectionOf>
</ObjectAllValuesFrom>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectAllValuesFrom>
<ObjectProperty URI="&ns;P1"/>
<OWLClass URI="&ns;C9"/>
</ObjectAllValuesFrom>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectHasValue>
<ObjectProperty URI="&ns;P1"/>
<Individual URI="&ns;I4"/>
</ObjectHasValue>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>
```

```
<SubClassOf>
<ObjectExistsSelf>
<ObjectProperty URI="&ns;P1"/>
</ObjectExistsSelf>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectExistsSelf>
<ObjectProperty URI="&ns;P1"/>
</ObjectExistsSelf>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectExactCardinality cardinality="3">
<ObjectProperty URI="&ns;P1"/>
<ObjectUnionOf>
<ObjectExactCardinality cardinality="2">
<ObjectProperty URI="&ns;P0"/>
<ObjectAllValuesFrom>
<ObjectProperty URI="&ns;P0"/>
<ObjectOneOf>
<Individual URI="&ns;I0"/>
<Individual URI="&ns;I3"/>
</ObjectOneOf>
</ObjectAllValuesFrom>
</ObjectExactCardinality>
<OWLClass URI="&ns;C29"/>
<OWLClass URI="&ns;C23"/>
<ObjectSomeValuesFrom>
<ObjectProperty URI="&ns;P1"/>
<OWLClass URI="&ns;C8"/>
</ObjectSomeValuesFrom>
</ObjectUnionOf>
</ObjectExactCardinality>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>
```

```
<SubClassOf>
<ObjectMinCardinality cardinality="1">
<ObjectProperty URI="&ns;P0"/>
<ObjectOneOf>
<Individual URI="&ns;I0"/>
<Individual URI="&ns;I3"/>
</ObjectOneOf>
</ObjectMinCardinality>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectMaxCardinality cardinality="2">
<ObjectProperty URI="&ns;P0"/>
<ObjectOneOf>
<Individual URI="&ns;I4"/>
<Individual URI="&ns;I2"/>
</ObjectOneOf>
</ObjectMaxCardinality>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectExactCardinality cardinality="2">
<ObjectProperty URI="&ns;P1"/>
</ObjectExactCardinality>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectExactCardinality cardinality="2">
<ObjectProperty URI="&ns;P1"/>
</ObjectExactCardinality>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
```

```
<ObjectMinCardinality cardinality="0">
<ObjectProperty URI="&ns;P0"/>
</ObjectMinCardinality>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

<SubClassOf>
<ObjectMaxCardinality cardinality="3">
<ObjectProperty URI="&ns;P1"/>
</ObjectMaxCardinality>
<OWLClass URI="&ns;Thing"/>
</SubClassOf>

</Ontology>
```

References

[1] International Telecommunication Union (ITU). *Key Global Telecom Indicators for the World Telecommunication Service Sector*. 2007. http://www.itu.int/ITU-D/ict/statistics/at_glance/KeyTelecom99.html.

[2] Netcraft. *April 2008 Web Server Survey*. Apr 14, 2008. http://news.netcraft.com/archives/web_server_survey.html.

[3] Ivan Herman. *W3C Semantic Web Activity*. Nov 18, 2007. http://www.w3.org/2001/sw/.

[4] D. L. McGuinness and F. van Harmelen. *W3C Rec OWL Web Ontology Language*. Feb 10, 2004. http://www.w3.org/TR/owl-features/.

[5] Mike Dean, Dan Connolly, Frank van Harmelen, James Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. PatelSchneider, and Lynn Andrea Stein. *OWL Web Ontology Language 1.0 Reference*. July 29, 2002. http://www.w3.org/TR/2002/WD-owl-ref-20020729/.

[6] Ian Horrocks, Bijan Parsia, Peter F. Patel-Schneider, and Ulrike Sattler. *OWL 1.1 Web Ontology Language*. December 19, 2006. http://www.w3.org/Submission/2006/SUBM-owl11-overview-20061219/.

[7] Ian Horrocks, Boris Motik, and Peter F. Patel-Schneider. *OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax*. April 11, 2008. http://www.w3.org/TR/2008/WD-owl2-syntax-20080411/.

[8] Dmitry Tsarkov and Ian Horrocks. *Reasoner Prototype*. September 25, 2003. http://wonderweb.semanticweb.org/deliverables/documents/D13.pdf.

[9] Li Ma, Yang Yang, Zhaoming Qiu, GuoTong Xie, Yue Pan, and Shengping Liu. *Towards a Complete OWL Ontology Benchmark*. In Sure and Domingue [14], pages 125–139. http://www.springerlink.com/content/l0wu543x26350462/fulltext.pdf.

[10] Timo Weithöner, Thorsten Liebig, Marko Luther, Sebastian Böhm, Friedrich W. von Henke, and Olaf Noppens. Real-world reasoning with owl. In Franconi et al. [15], pages 296–310. http://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.090/Publikationen/2007/WLLBvHN07.pdf.

[11] Boris Motik, Peter F. Patel-Schneider, and Ian Horrocks. *OWL 1.1 Web Ontology Language Structural Specification and Functional-Style Syntax*. May 23, 2007. http://www.webont.org/owl/1.1/owl_specification.html.

[12] Grady Booch, James E. Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[13] Ian Horrocks, Peter F. Patel-Schneider, Sean Bechhofer, and Dmitry Tsarkov. *OWL rules: A proposal and prototype implementation*. *J. Web Sem.*, 3(1):23–40, 2005. http://www.cs.man.ac.uk/~horrocks/Publications/download/2005/HPBT05.pdf.

[14] York Sure and John Domingue, editors. *The Semantic Web: Research and Applications, 3rd European Semantic Web Conference, ESWC 2006, Budva, Montenegro, June 11-14, 2006, Proceedings*, volume 4011 of *Lecture Notes in Computer Science*. Springer, 2006.

[15] Enrico Franconi, Michael Kifer, and Wolfgang May, editors. *The Semantic Web: Research and Applications, 4th European Semantic Web Conference, ESWC 2007, Innsbruck, Austria, June 3-7, 2007, Proceedings*, volume 4519 of *Lecture Notes in Computer Science*. Springer, 2007.

[16] Matthew Horridge, Sean Bechhofer, and Olaf Noppens. *Igniting the OWL 1.1 Touch Paper: The OWL API*. In Golbreich et al. [18]. http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-258/paper19.pdf.

[17] Natalia Villanueva-Rosales and Michel Dumontier. *Describing Chemical Functional Groups in OWL-DL for the Classification of Chemical Compounds*. In Golbreich et al. [18]. http://www.scs.carleton.ca/~nvillanu/papers/2007_OWLED_CFG.pdf.

[18] Christine Golbreich, Aditya Kalyanpur, and Bijan Parsia, editors. *Proceedings of the OWLED 2007 Workshop on OWL: Experiences and Directions, Innsbruck, Austria, June 6-7, 2007*, volume 258 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[19] Jian Zhou, Li Ma, Qiaoling Liu, Lei Zhang, Yong Yu, and Yue Pan. *Minerva: A Scalable OWL Ontology Storage and Inference System*. In Mizoguchi et al. [20], pages 429–443. `http://dx.doi.org/10.1007/11836025_42`.

[20] Riichiro Mizoguchi, Zhongzhi Shi, and Fausto Giunchiglia, editors. *The Semantic Web - ASWC 2006, First Asian Semantic Web Conference, Beijing, China, September 3-7, 2006, Proceedings*, volume 4185 of *Lecture Notes in Computer Science*. Springer, 2006.

[21] Boris Motik and Ulrike Sattler. *A Comparison of Reasoning Techniques for Querying Large Description Logic ABoxes*. In Hermann and Voronkov [22], pages 227–241. `http://dx.doi.org/10.1007/11916277_16`.

[22] Miki Hermann and Andrei Voronkov, editors. *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, volume 4246 of *Lecture Notes in Computer Science*. Springer, 2006.

[23] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. *OWLIM - A Pragmatic Semantic Repository for OWL*. In Dean et al. [24], pages 182–192. `http://dx.doi.org/10.1007/11581116_19`.

[24] Mike Dean, Yuanbo Guo, Woochun Jun, Roland Kaschek, Shonali Krishnaswamy, Zhengxiang Pan, and Quan Z. Sheng, editors. *Web Information Systems Engineering - WISE 2005 Workshops, WISE 2005 International Workshops, New York, NY, USA, November 20-22, 2005, Proceedings*, volume 3807 of *Lecture Notes in Computer Science*. Springer, 2005.

[25] Thorsten Liebig. *Reasoning with OWL, System Support and Insights*. September, 2006. `http://www.informatik.uni-ulm.de/ki/Liebig/papers/TR-U-Ulm-2006-04.pdf`.

[26] Timo Weithner, Thorsten Liebig, Marko Luther, and Sebastian Bhm. *What's Wrong with OWL Benchmarks?* November 2006. http://www.informatik.uni-ulm.de/ki/Liebig/papers/weithoener-et-al-ssws06.pdf.

[27] Ivan Herman and Sandro Hawke. *OWL Working Group Charter*. September 7, 2007. http://www.informatik.uni-ulm.de/ki/Liebig/papers/weithoener-et-al-ssws06.pdf.