CQChecker: A Tool to Check Ontologies in OWL-DL using Competency Questions written in Controlled Natural Language

[1,2]Camila Bezerra
[2] Filipe Santana
[2] Fred Freitas

[1]Federal University of Recôncavo in Bahia
Center for Exact Sciences and Technology
e-mail:camilabezerra@ufrb.edu.br
Cruz das Almas – Bahia - Brazil

[2]Federal University of Pernambuco
Informatics Center
e-mails: {*fss3,fred*}@cin.sufpe.br
Recife – Pernambuco - Brazil

Abstract. Competency Questions (CQs) play an important role in ontology development lifecycles as they represent functional ontology requirements. One of the main problems that hamper their proper use lies in the lack of tools that assist users to check if CQs are being fulfilled by the ontology being defined, particularly when these ontologies are defined in OWL (Ontology Web Language) under Description Logic formalism. Recently there has been a trend in checking CQs against ontologies using the RDF query engine SPARQL. Naturally, this language, being created for the formalism of Semantic Networks, is clearly not expressive enough, and, thus, inadequate to check the fulfillment of OWL CQs. As SPARQL queries can be performed only at the assertional level (instances), or at most the schema level, they are not shaped to entail an answer which may be deduced by the ontology using a subsumption not explicit in the ontology. The tool takes advantage of the WordNet lexical to deal with synonyms and adjectives stated in the CQs. In some cases, the tool shows an explanation of why the CQ being treated is considered valid with regard to the ontology. We present the tool's architecture, capabilities and test examples against a number of controlled natural language CQs.

Keywords- competency questions, checking, ontology.

1 Introduction

Due to the recent growth of the semantic web, several methodologies and tools have appeared to support ontology development. The ontology development lifecycle shares some similarities and differences from the software development lifecycle.

Software requirements specifications consist of a description of desirable functionalities and behavior of a system. Hofmann and colleagues [1] affirm that finding and fixing a software problem after delivery is 100 times more expensive than during the requirements and early design phases. This means that contradictions, negligence, ambiguity, not previously discovered, will bring undesirable consequences. Such a statement also applies to ontology engineering. An ontology poorly specified probably will not meet the ontology engineer's desired objectives. In addition, it may provoke overwork, and thus, higher maintenance costs.

Most ontology development methodologies ([2], [3], [4]) define the ontology requirements as Competency Questions (CQs). These consist of a set of questions stated and answered in natural language, so that the ontology must be able to answer them correctly [5]. However, traditional methodologies have treated the use of Competency Questions (CQs) superficially. For instance, the process of checking whether they are fulfilled by the developed ontology via reasoning is not well documented, nor automated.

[6] introduced the idea of how the ontology engineer should describe a CQ, by means of axioms and text. However, CQs can be translated into Ontology Web Language (OWL) [7] which implements Description Logics (DL) formalism. Recently there has been a trend to check OWL ontologies using DL [8]; and with the RDF query engine SPARQL [9]. However, being created for the formalism of Semantic Nets [10], SPARQL is not expressive enough to describe and check the fulfillment of CQs in OWL. This is due to the fact that SPARQL was conceived as a query language to retrieve data from RDF models; therefore, it can be used only at the assertional level (instances) [11], or at most at the schema level. SPARQL indeed plays an important role in checking CQs for RDF ontologies; however, to entail an answer which is not in the ontology but that can be subsumed by DL reasoning is certainly beyond its capabilities.

With such things in mind, in this work we propose a tool called CQChecker to support the automation of checking CQs during Ontology Evaluation, and specifically for asserting functional requirements expressed as CQs. Furthermore, CQChecker

provides a mechanism to enable us to verify whether or not the ontology meets its corresponding CQs by supporting both assertional and terminological queries. To accomplish that, we created a modular architecture where each module treats a different CQ type. CQChecker uses WordNet to deal with synonyms and adjectives, frequently found in non-controlled natural language CQs.

This paper is organized as follows: section 2 describes the main concepts about Competency Questions and Ontology Engineering; section 3 presents our tool, the CQChecker; section 4 presents the results obtained so far; section 5 is devoted to related work; and section 6 presents some conclusions and future work.

2 Competency Questions and Ontology Engineering

The term ontology originates from Philosophy and is related to the study of existence. In Computer Science, Gruber [12] described that an "ontology is an explicit specification of a conceptualization". In this definition, the term conceptualization corresponds to the concepts and other entities that must exist in a domain of interest, as well as the relationships between them. In Computer Science, (formal) ontologies are generally used as a way to represent domain knowledge, support semantic interoperability, and provide automated reasoning about domain knowledge, among other uses [10].

In order to enable the creation of ontologies with a certain degree of organization and documentation, there are several ontology engineering approaches. We describe the main ones in the following items.

- Uschold and Gruninger [2]: Describes the documentation process, the development and evaluation of an ontology from informal to formal techniques. Moreover, it takes into account the formal representation of CQ in a way that recognizes axioms for particular ontology descriptions.
- Methontology [3]: Aims at elucidating the activities necessary to create new ontologies. This methodology describes the ontology development process by splitting it into types of activities. The most important step is the specification, which supports the creation of an ontology specification document, written in a controlled natural language and using CQ.
- NeOn Project methodology [4]: Emerged as a proposal from an evolution of the already available methods and includes the collaborative and distributed development. Like other methodologies, this one also introduces the idea of CQ as a viable choice for identification of ontology requirements.

However, most methodologies describe CQs superficially. CQs are mentioned as the initial documentation phase, supported by controlled natural language-based statements [8]. Unfortunately, there is a lack of mechanisms to evaluate to what extent the ontology fulfills the CQ. This is what the current approach relies upon.

Considering OWL, one of the standards for representing ontologies, and in order to define our approach formally, a Description Logic Ontology (henceforth, an ontology) can be described as a set of axioms defined by the triple (N_C, N_R, N_O) [13], where:

- N_C is the set of concept names or atomic concepts (unary predicate symbols),
- N_R is the set of role or property names (binary predicate symbols),
- N_O is the set of individual names (constants), instances of N_C and N_R

In Table 1, we use Tarski-style semantics to define the possible DL constructs we may work with. Assuming that C and D are arbitrary concept descriptions; $r, s \in N_R$, $a, b \in N_O$ and $\#r^I(x, C)$ denote the cardinality of $\{y \mid (x, y) \in r^I, y \in C^I\}$. A concept description C is said to be *satisfiable* by the interpretation I iff $C^I \neq \emptyset$.

There are two axiom types allowed in DL:

(i) Assertional axioms, which are concept assertions $C(a)$, or role assertions $R(a,b)$, where $C \in N_C$, $r \in N_R$, $a, b \in N_O$. And,

(ii) Terminological axioms, in one of the forms $C \sqsubseteq D$ or $C \equiv D$, the latter standing for $C \sqsubseteq D$ and $D \sqsubseteq C$, C and D being concept expressions.

$\mathcal{T}$ is satisfiable by interpretation I if and only if I satisfies all axioms in $\mathcal{T}$. An Abox $\mathcal{A}$ with regard to a Tbox $\mathcal{T}$ is a finite set of assertions of the form $C(a)$, $r(a,b)$, $a=b$ and $a \neq b \in N_O$. An Abox $\mathcal{A}$ is satisfiable by an interpretation I if I satisfies $\mathcal{T}$ and all assertions in $\mathcal{A}$.

Different DL fragments (or languages) can be built from the constructs listed in Table 1. For instance, $\mathcal{EL}$ comprises the set of constructs $\{A, C \sqcap D, \exists r.C\}$; $\mathcal{ALC}$ adds to $\mathcal{EL}\{\sqcup, \forall r.C, \neg\}$; $\mathcal{ALCN}$ extends $\mathcal{ALC}$ with $\leq n\, r$ and $\geq n\, r$, while $\mathcal{ALCQ}$ encompasses all constructs from the table [11].

.

Table 1: Syntax and semantics of SHQ

Syntax	Semantics
A	$A^I \subseteq \Delta^I, A \in N_C$
$r \in N_R$	$r^I \subseteq \Delta^I \times \Delta^I$
$C \sqsubseteq D$	$C^I \subseteq D^I$
$C(a)$	$a^I \in C^I$
$r(a,b)$	$(a,b)^I \in r^I$
$\neg C$	$\Delta^I \backslash C^I$
$C \sqcap D$	$C^I \cap D^I$
$C \sqcup D$	$C^I \cup D^I$
$\exists r.C$	$\{x \mid \exists y\ (x,y) \in r^I \wedge y \in C^I\}$
$\forall r.C$	$\{x \mid \forall y\ (x,y) \in r^I \Rightarrow y \in C^I\}$
$\geq n\ r$	$\{x \mid \#r^I(x,y) \geq n\}$
$\leq n\ r$	$\{x \mid \#r^I(x,y) \leq n\}$
$\geq\ n\ r.C$	$\{x \mid \#r^I(x,C) \geq n\}$
$\leq\ n\ r.C$	$\{x \mid \#r^I(x,C) \leq n\}$

In the next section, we define and discuss competency questions, their practice and current shortcomings regarding the way they are put into practice.

2.1 Competency Questions

Given a set of scenarios related to a domain of discourse, developers should be able to describe a set of questions which aim to represent user demands and scope limits. These questions are usually written in controlled natural language and support the development process in two ways:

- By enabling the identification of CQ core elements (classes and instances) and their relationships in order to create the ontology vocabulary, and;
- By providing a simple means to verify requirements' satisfiability either by knowledge retrieval or entailment through their axioms.

To the best of our knowledge, the latter – namely entailment through axioms – is not sufficiently seen to by the main ontology engineering methodologies. For instance, NeOn [4] suggests checking CQ satisfiability in a manual way. In turn, SPARQL which is the most used tool for checking CQs in an automated way, does not perform DL reasoning, and therefore is not capable of proving a CQ answer that can only satisfied by DL entailment.

Noy and Hafner [5] believe that, in order to define the domain and scope of an ontology during the specification phase, the ontology being built should be based on the answers to questions like [5]:

- Which domain will the ontology cover?
- For what purposes, queries and uses is the ontology being designed?
- For what types of questions should the ontology provide answers?
- Who will use and maintain the ontology?

The answers to these questions may change during ontology development, when the ontology engineer requires more detailed information from the user to describe the domain. For instance, when new information emerges about the domain as new concepts and relations, or when it is necessary to describe some information in more detail.

In the next section, we define what a formal competency question is.

2.2 Formal Competency Questions

Although we are dealing with ontologies written in DL language, we prefer to define what a formalized competency question is independent of particular formalisms and languages. We only assume that there exist formal languages for expressing ontologies and queries. These languages and formalisms might enable queries with bound queries/answers, where this variable tuple can be empty.

The ontology formalization, thus, defines vocabularies and logical consequences. Below, our definitions:

- $Voc(O)$ is the vocabulary, *i.e.*, defined and used in ontology *O*.

- We denote by $O \vDash \delta$ the fact δ is true in all the possible models of *O*. We first consider a competency question as a pair composed of a question and its answer.

Definition 1 (Competency Questions). *A competency question is pair* $\langle Q, \sigma \rangle$ *such that* Q *is a query expressed in a formal language and* σ *is an answer to this query expressed as a variable substitution.*

Definition 2 (Satisfied competency question). *A competency question* $\langle Q, \sigma \rangle$ *is satisfied by an ontology O if* $O \models \sigma(Q)$.

In the next section, we present our current work, a tool to check whether a CQ Q is being met by ontology O.

3 CQChecker: a tool to check competency questions on ontologies

CQChecker is modularly structured (Figure 1). The layer UTILITY represents the APIs used in the tool. We use OWL API [14] in all three modules of the tool to extract information as axioms and classes of the ontology. OWL API [14] is a Java interface and implementation for OWL.

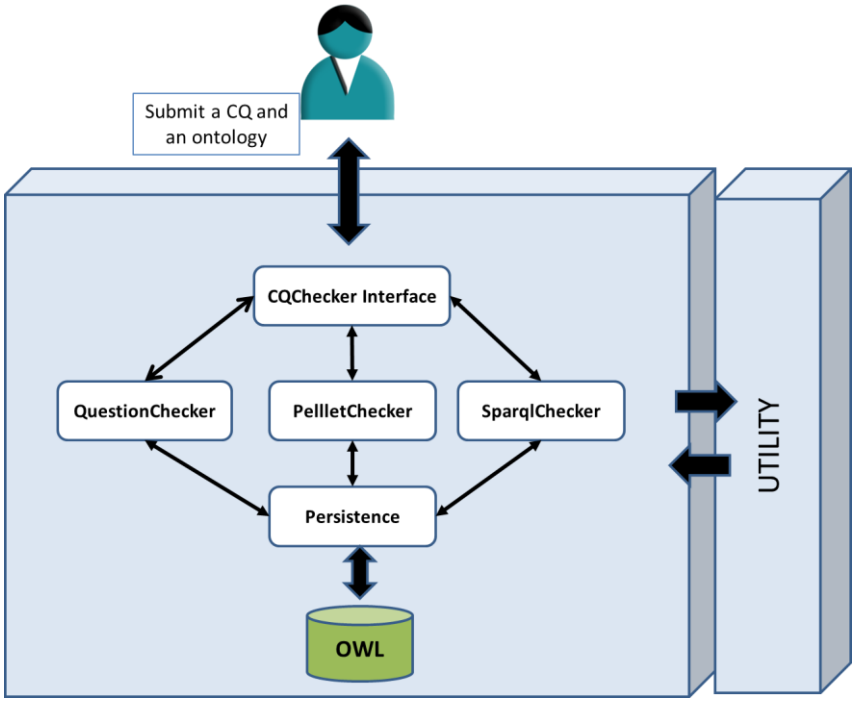

Figure 1: Architecture of CQChecker

The user submitted the CQ and the ontology, and the CQ will be analyzed and directed to one of three modules: QuestionChecker, PellletChecker or SparqlChecker. Each module treats a distinct kind of CQ expressed in controlled natural language, as specified in the next subsection.

The basic functioning of the tool can be summarized in the following terms: first, it analyzes the CQ in order to classify it into one of three types, according to the possible answer it is supposed to retrieve (over classes or instances). Then, the system directs the CQ to the corresponding module, where it will be converted and checked.

To reach a better understanding in the next sections, first we will explain the ontology which is used as a reference to describe the process and examples presented in the next sections.

3.1 Pizza Ontology

The Pizza ontology describes the pizza domain and is available at Protégé website [15]. Figure 2 briefly introduces the Pizza Ontology.

The purple arrow denotes superclass/subclass dependencies. Thus, the class *DomainConcept* has *Food* and *Country* as subclasses. On the other hand, *PizzaTopping*, *Pizza* and *PizzaBase* are subclasses of *Food*.

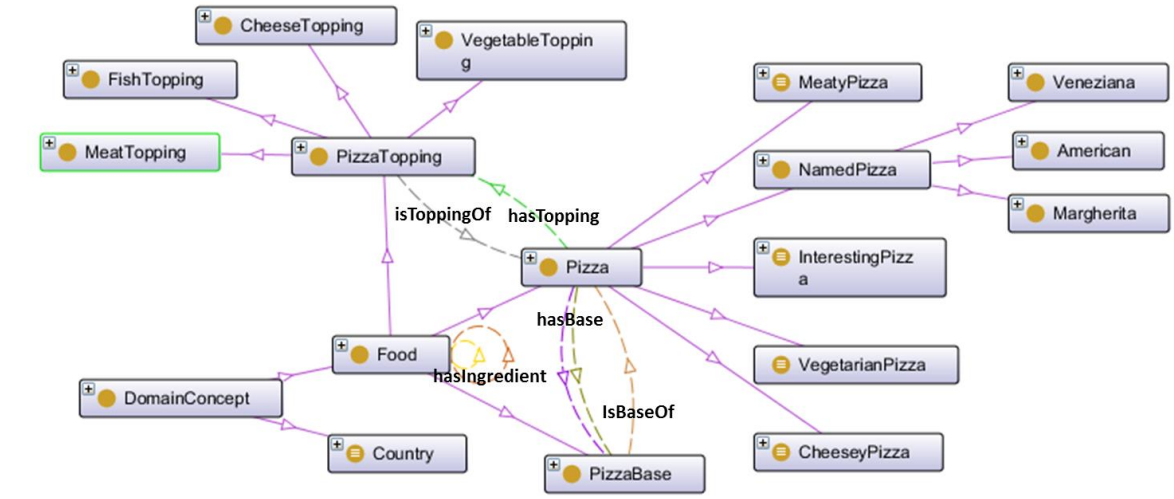

Figure 2: Pizza Ontology

The other relations are explained below:

$Pizza \sqsubseteq Food$

$PizzaTopping \sqsubseteq$ Food

$Pizza \sqsubseteq \exists hasBase.PizzaBase$

$Country \equiv \{America, Italy, Germany, France, England\}$

$CheeseyPizza \equiv Pizza \sqcap \exists hasTopping.CheeseTopping$

$MeatPizza \equiv Pizza \sqcap \exists hasTopping.MeatTopping$

$VegetarianPizza \equiv Pizza \sqcap \neg(\exists hasTopping.FishTopping) \cap \neg(\exists hasTopping.MeatTopping)$

$InterestingPizza \equiv Pizza \sqcap \leq 3 hasTopping$

$American \sqsubseteq NamedPizza \sqcap \exists hasCountryOfOrigin.\{America\}$

The class "*Country*" is equivalent to a set of individuals which are America, Italy, Germany, France and England. This means that Countries can only be either America, Italy, Germany, France or England.

The *CheesePizza* is a pizza that has some topping of cheese, and *MeatPizza* is a pizza that has some topping of meat.

The VegetarianPizza is a pizza that has neither fish nor meat.

For a pizza to be considered interesting, it needs to have at least three toppings. Finally, the *American* pizza is a *NamedPizza* and originates in America.

Before describing how the CQs are analyzed, it is important to clarify the possible CQ types.

3.2 Types of Competency Questions

In the current implementation, we consider CQs expressed as interrogative sentences. The difference among the CQ types is related to the type of answer retrieved by the CQ. We identified three main types:

1) *CQs which work over classes and their relations.* Here, the answer consists either of classes or of instances. For example, for the CQ "*Which are the Toppings of the Cheesey Pizza*?", the class "*CheeseTopping*" will return according to the Pizza ontology.

2) *Decision problems expressed as CQs*. In this type, the answer permitted to the question can only be true or false. Sometimes for solving CQs from this type, an inference process is necessary and, if the answer is true, an explanation

should be returned. For instance, for the CQ "*Is pizza a food?*", a "yes" will be the response and there will be an explanation for this, according to the Pizza ontology.

3) *CQs are expressed in an interrogative form which works only over instances*. For instance, the CQ "*Which are the possible countries of a pizza?"* gets as answer *"France, England, Italy, Germany, America*", according to the Pizza ontology.

Since we are dealing with controlled natural language, we are dealing with some patterns of CQs that we have found, shown in Table 2.

Table 2: Competency Questions Patterns

Pattern	Example	Ontology entities found in CQ
What + <property> +<class>?	What is the spiciness of chicken topping?	<property> = *hasSpiciness* <class> = *ChickenTopping*
How many + <property> + <class>?	How many toppings does the interesting pizza have?	<property> = *hasTopping* <class> = *InterestingPizza*
From which + <property> + <class>?	From which nation is American pizza?	<property> = *hasCountryOfOrigin* <class> = *American*
Is + <class>+ a +<class>?	Is pizza a food?	<class> = *Pizza* <class> = *Food*
Is + <individual>+ a +<class>?	Is Marguerita a vegetarian pizza?	<individual> = *Marguerita* <class> = *VegetarianPizza*
Which + type + <class>?	Which are the types of Pizza?	<class> = *Pizza*
Does <class> + <property> + <class>?	Does American pizza have mozzarella topping?	<class> = *American* <class> = *MozarellaTopping*
Which + disjoint <class>?	Which are the disjoint classes of Vegetarian Pizza?	<class> = *VegetarianPizza*
Are + <class> + <class> disjoint?	Are vegetarian pizza and non-vegetarian pizza disjoint?	<class> = *VegetarianPizza* <class> = *NonVegetarianPizza*
Which + <class> + <property> + <class> + not + <property> + <class>?	Which are the pizzas that have mozzarella topping but not have meat topping?	<class> = *Pizza* <property> = *hasTopping* <class> = *MozzarellaTopping* <class> = *MeatTopping*
Is <class> + <class> + or + <class>?	Is capriciosa a pizza or an ice cream?	<class> = *Capriciosa* <class> = *Pizza* <class> = *IceCream*
Is <class> + <individual> + or + <individual>?	Is Medoc red or white?	<class> = *Medoc* <individual> = *Red* <individual> = *White*
Which + instances + <country>?	Which are the instances of the country?	<class> = *Country*
Is <individual> + <class>?	Is France the type of Country?	<individual> = *France* <class> = *Country*

In the following section, we describe each module of the system.

3.3 QuestionChecker

This module works with CQ type 1. It evaluates whether the CQ is met by a given ontology. As previously stated, the CQ tests performed by this module are related to the terminological level.

To accomplish this, the algorithm displayed in Figure 4 should be run. Basically, it takes a CQ, splits it into tokens and tries to find the concepts and relations from the ontology described in OWL DL, which the CQ referred to. Consider the CQ "**Which are the Toppings of the CheeseyPizza?**" about the Pizza ontology. With it, we want to check if there is any class that would provide an instance for the image of *CheeseyPizza*, through the relation *hasTopping*, which is a *Topping*. For this to be achieved, we first search for a relation which can be a verb or a noun, and afterwards we look for the concept, which has an image that is a Topping in the relation, in this case *CheeseyPizza*. The algorithm will perform the following steps:

1. *Split the CQ into a list of tokens.*
    a. In this case, the tokens are "Which", "are", "the", "Toppings", "of", "the", "*Cheesey" and "Pizza"*
2. *Find in the given CQ a main relation (property) to which the main class is referred.*
    a. In this case, the relation is *hasTopping*, because "Toppings" is the only token similar to a property of the ontology. To obtain this, we use a similarity algorithm.
    b. The token "toppings" is deleted from the list of tokens, because "toppings" was already used to find the relation.
3. *Find among the tokens a main class (concept) which the CQ deals with.*
    a. In this case, the main class is *CheeseyPizza*, because "Cheesey" and "Pizza" are the only tokens left which, together, form a word that corresponds to a class in the ontology.
4. *Get from the ontology the axioms that contain the main concept.*
    a. In this case, the system will return the axiom (in OWL):

ObjectSomeValuesFrom (http://www.co-ode.org/ontologies/pizza/pizza.owl#hasTopping , pizza.owl#CheeseTopping)

5. *Select what axioms contain the main concept and main property.*
    a. The only axiom of the previous step is thus selected.
6. *Identify what the type of axiom is.*
    a. In this case, the type is $\exists P.C$, i.e, an existential restriction.
7. *Identify and return the concepts or individuals from the axioms that correspond to the answer to the CQ.*
    a. The returned answer is *CheeseTopping*.

The example is portrayed, for the sake of readability, in Figure 3.

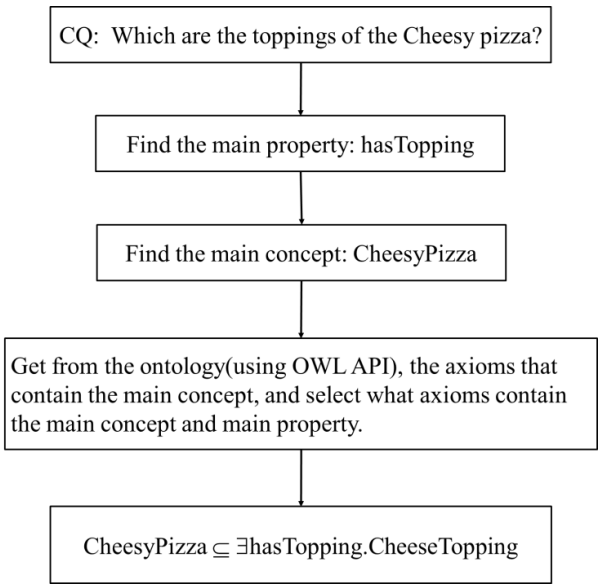

Figure 3: Example of the CQChecker algorithm.

Figure 4 displays the detailed, formalized algorithm. The inputs of the algorithm are: CQ in controlled natural language; and, the ontology to be checked using its respective CQ. As output the algorithm returns $\propto |O \models \propto (CQ_{DL})$. To accomplish that, the CQ is broken into tokens and stored at "*TokenSet*" variable. After that, the algorithm searches the main object properties and the main classes among the tokens. Finally, it gets all axioms that contain the main object property and the main classes, identifying the types of these axioms and displaying all classes or individuals that are part of the selected axioms.

Step 6 of the algorithm can be detailed in the following steps:

1) The algorithm searches for the main concept compared with the class names of the ontology concepts, ignoring upper case and lower case letters.
2) First it verifies if the last word is a class. If yes, it asks the user if this word is the main concept of the CQ. If the user confirms this, then the last word will be the MainConceptToken. Otherwise, it will try the next step.
3) Afterwards, it verifies if the penultimate word is a class. If yes, it asks the user if this word is the main concept of the CQ. If the user confirms, then the last word will be the MainConceptToken. Otherwise, it will try the next step.
4) It is tested if the last two tokens and the last three tokens of the CQ together consist of a class of the ontology. If yes, it asks the user if this word is the main concept of the CQ. If the user confirms, the last two tokens will be considered as the MainConceptToken of the CQ.
5) If even after the previous tests the main concept is not found, then the WordNet API is used to get the synonyms of the last words, and test if one of them is a class. If yes, this class will be considered the MainConceptToken.

Question Checker

Inputs: CQ_{NL} (Competency questions in controlled natural language),

O = set of axioms a_i defined over the triple (N_C, N_R, N_O) [13], where

$-N_C$ is the set of *concept names*,

- N_R is the set of *role or property names*,

- N_I the set of *instances* of N_C and N_R:

Output: $\propto \mid O \models \propto (CQ_{DL})$, where CQ_{DL} is a competency question written in DL

1. TokenSet := {token | token extracted from CQ_{NL}}
2. MainRoletToken := token[1]
3. **For** i=1 **to** size(TokenSet)
    a. **For** j=1 **to** size(N_R)
        i. **If** token[i] = r[j] ∈ N_R **then**
            1. MainRoleToken := r[j]
            2. **Break**
        ii. **Endif**
    b. **Endfor**
4. **Endfor**
5. MainConceptToken := token[1]
6. **For** i=1 **to** size(TokenSet)
    a. **For** j=1 **to** size(N_C)
        i. **If** token[i] = C[j] ∈ N_C **then**
            1. MainConceptToken := C[j]
            2. **Break**
        ii. **Endif**
    b. **Endfor**
7. **Endfor**
8. AxiomSet = {a' ∈ a_i | MainConceptToken ∈$_{lexically}$ a and MainRoleToken ∈$_{lexically}$ a}
9. $CQ_{DL} := \wedge_{i=1}^{n} a'_i \in$ AxiomSet
10. **return** $\propto \mid O \models \propto (CQ_{DL})$

Figure 4: Algorithm

We use WordNet to support CQs in controlled natural language with the aim of finding the classes from the ontology. WordNet [16] is a lexical database of English. It groups English words, like, nouns and verbs, in a set of synonyms, called synsets. To exemplify the steps above, consider Figures 5 e 6.

In Figure 5, suppose that the user enters a CQ but he does not know the real name of the classes or of the properties, *i.e*, he will use synonyms to reference them. For instance, considering the CQ "***From which nation is American pizza*?**", the word "*nation*" does not correspond to any class in the ontology. In this case, the QuestionChecker gets the synonyms of "*nation*" and tests whether or not one of these synonyms is similar to a name of some property in the ontology. In this case, it will find out that the synonym "country" is similar to the property "hasCountryOfOrigin**"** in the ontology, then it will confirm if "hasCountryOfOrigin**"** is the main relation by asking the user; and it executes the rest of the algorithm using the property

"hasCountryOfOrigin". After that, it will search the main concept, and then it will test if "*American*" is a class and ask the user if "*American*" is the main concept.

Another feature consists of the fact that the QuestionChecker is able to treat cases where the last two tokens of the CQ are the main concept from the CQ*, i.e*, they are treated as a single class. For instance, consider the CQ "**How many toppings does the Interesting Pizza have?**" in Figure 6. The two last words "interesting" and "pizza" together comprise the name of a single class named "*InterestingPizza*" in the pizza ontology, which is the main class that the CQ talks about.

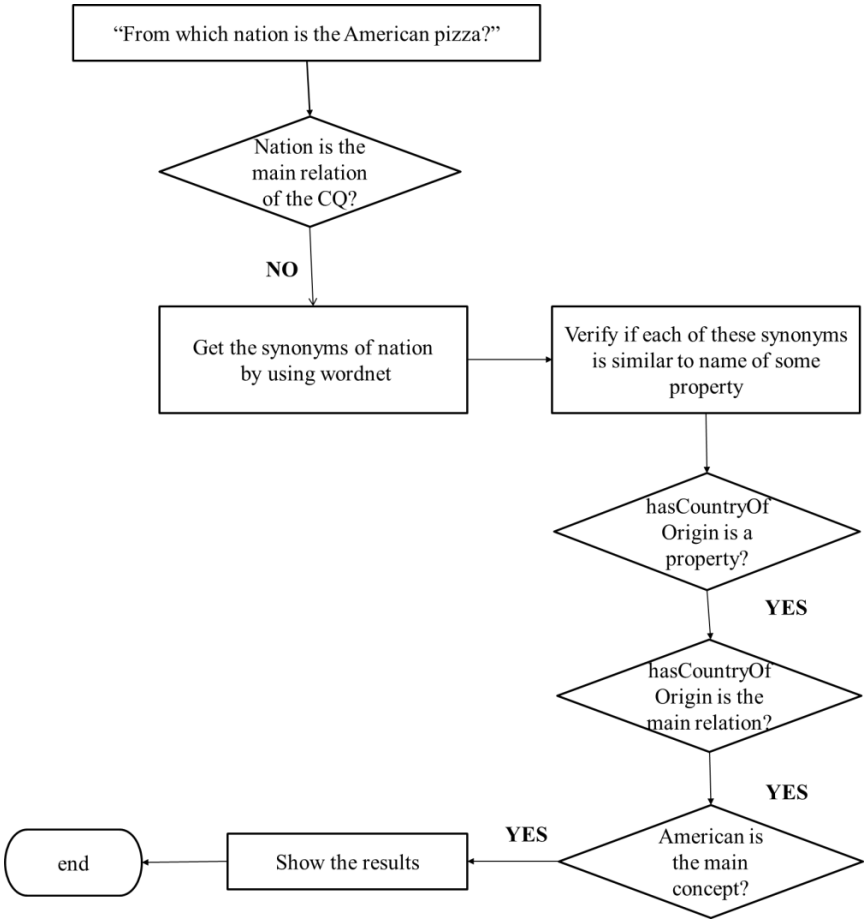

Figure 5: Example

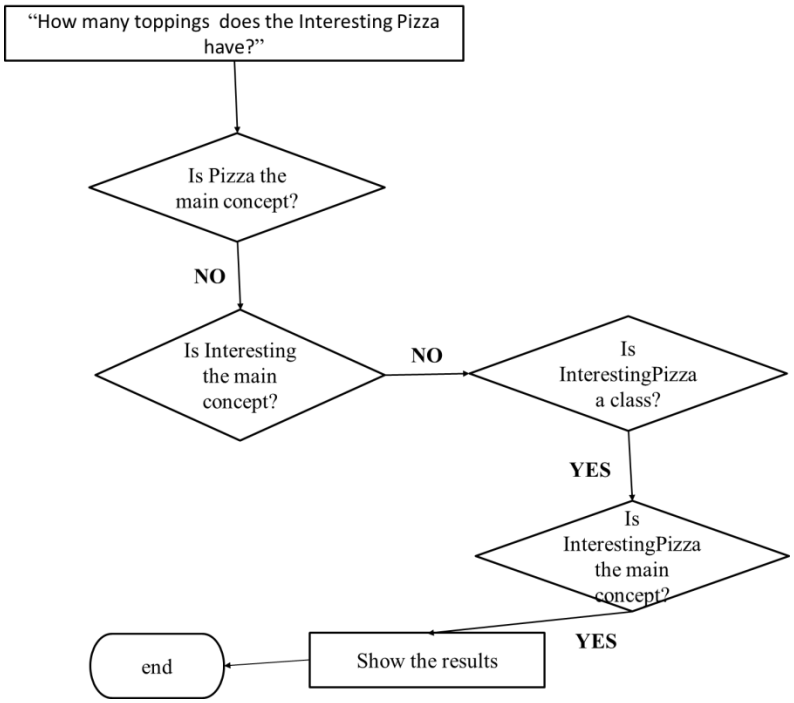

Figure 6: Example

3.4 InferenceChecker

This module deals with CQs type two. It receives a CQ which expresses an axiom that the user wishes to know is true or false; this axiom is proven to be valid or not against the ontology. In the case of a true statement, it shows an explanation of why this CQ is valid in relation to the ontology. Here, we are treating axioms of class hierarchies, subsumption, and disjointness. To implement this, we relied on the Pellet reasoner [17]. This reasoner allows to test whether or not a class is a subclass of another class. Another possible usage for it is to check ontology consistency. We use Pellet to check if a CQ is being fulfilled, and true or false is returned as a response. Moreover an explanation is shown in the case that the result is true.

To find the main classes and relations of the CQ, we build a similar algorithm for QuestionChecker, shown in section 4.2.

In this module, we focus on DL queries that require reasoning related to hierarchy among classes (and, as a consequence, if an instance is a member of a class). For instance, consider the description about pizzas below:

- Margherita is a Pizza and has topping of Mozzarela or Tomato.
- Vegetarian Pizzas are the Pizzas that do not have a topping of fish or meat.
- Mozzarela is not fish or meat.

A CQ about this description could be "***Is* Margherita *a Vegetarian Pizza*?**". A DL reasoner summons as true. For the latter one, the InferenceChecker will return the following explanation, which confirms that Marguerita is a Vegetarian Pizza:

1) Margherita subClassOf hasTopping only MozzarellaTopping or TomatoTopping *MozzarellaTopping subClassOf CheeseTopping* *TomatoTopping subClassOf VegetableTopping* *CheeseTopping disjointWith MeatTopping* *FishTopping disjointWith VegetableTopping* *Margherita subClassOf NamedPizza*	$Marguerita \sqsubseteq \forall hasTopping.MozzarellaTopping \cup \forall hasTopping.TomatoTopping$ $MozzarellaTopping \sqsubseteq CheeseTopppping$ $TomatoTopping \sqsubseteq VegetableTopppping$ $ChesseTopping \sqsubseteq \neg MeatTopping$ $FishTopping \sqsubseteq \neg VegetableTopping$ $Margherita \sqsubseteq NamedPizza$
2) VegetarianPizza equivalentTo Pizza and not hasTopping some FishTopping and not hasTopping some MeatTopping *CheeseTopping disjointWith FishTopping* *MeatTopping disjointWith VegetableTopping* *NamedPizza subClassOf Pizza*	$VegetarianPizza \equiv Pizza \sqcap \neg(\exists hasTopping.FishTopping) \cap \neg(\exists hasTopping.MeatTopping)$ $CheseeTopping \sqsubseteq \neg FishTopping$ $MeatTopping \sqsubseteq \neg VegetableTopping$ $NamedPizza \sqsubseteq Pizza$

3.5 SPARQLChecker

This module solves CQs type three. It receives a CQ expressed in interrogative form and the answer consists of instances, to which the CQ refers. Here, we are considering questions about individuals and class hierarchy; no explanation is provided here.

SPARQL [9] queries consist of triple patterns, *e.g.* < *subject*, *predicate*, *object* > format. We chose SPARQL to query ontologies, since this query language is considered a robust, consolidated language for this type of task. With SPARQL, we can create queries over instances using operations like filtering, difference, union, among others. Considering CQChecker, SPARQL is included in this module to treat CQs about instances and class hierarchies. For instance, the CQs "***Which are the instances of the Country?***" is type three, so it will be treated by the SPARQLChecker.

To this end, the SPARQLChecker has a similar algorithm to the QuestionChecker, where the algorithm gets the key words which are necessary for the query using SPARQL. After finding the key words, the algorithm creates the query in SPARQL. The result of this query is displayed to the user.

4 Results

We made three rounds of testing, each one with a different ontology. For each round ten CQs were used. The used ontologies are available on the Protégé website [15]. In these tests, the CQs were created by the development team, and we believe that the chosen CQs are representative for this first moment.

The CQs are related to class hierarchies, individuals, disjoint classes, intersection and union of classes; equivalent classes, universal and existential quantification; "has-value" restriction, and cardinality restriction.

The goal of these tests is to check if the ontologies were correctly specified according to a set of CQs.

4.1 First test: Wine ontology

First we tested ten CQs for the wine ontology. The CQs are displayed in Table 3.

Table 3: First Test

Number	Competency Question	Answer	Correct?
1	*What is the color of Cabernet Franc?*	Red	Yes
2	*What is the flavor of Cabernet Franc?*	Moderate	Yes
3	*Which are the types of Bordeaux Wine?*	RedBordeaux, StEmilion, Sauterne WhiteBordeaux, Medoc	Yes
4	*How many colors does a wine have?*	1	Yes
5	*How many flavors does a wine have?*	1	Yes
6	*Are LateHarvest and EarlyHarvest disjoint?*	Yes	Yes
7	*Which are the instances of PinotNoir?*	MountEdenVineyardEstatePinotNoir, MountadamPinotNoir, LaneTannerPinotNoir	Yes
8	*What is the sugar of Beaujolais?*	Dry	Yes
9	*What is the body of the Merlot Wine?*	Medium, Light	Yes
10	*What is the location of French Wine?*	FrenchRegion	Yes

CQs 1 and 2 were created to retrieve information concerning the *Cabernet Franc* wine. With these queries, we want to validate if the created CQs are met by the ontology, regarding color and flavor. For the 1st query, CQChecker will return "red", that is, the instance of the class *WineColor*, as the answer of CQ 1. And "moderate", the instance of the class *WineBody*, as the answer of CQ 2.

CQ 3 is related to the Bordeaux wine. We want to retrieve all subclasses of Bordeaux wine, in this case, *RedBordeaux*, *StEmilion*, *Sauterne*, *WhiteBordeaux*, and *Medoc*.

CQs 4 and 5 are generic questions, as they deal with the superclass Wine, regardless of specific wine types. In CQ 4, we want to check how many colors a wine can have. In CQ number 5, we want to know how many flavors a wine can have. In both cases, the answer is one.

In CQ 6, we want to check if the *Late Harvest* and *Early Harvest* are indeed disjoint.

In CQ 7, we want to know what the instances of *PinotNoir* wine are.

CQ 8 is about the *Beaujolais* wine. We want to check its sugar, which is dry- an instance of *WineSugar*.

CQ 9 is about the *Merlot* wine. We want to check its body, which could be medium or light, instances of *WineBody*.

And finally, CQ number 10 checks the location of French wine.

4.2 Second test: Travel ontology

In the second round of tests, we used the Travel ontology. The used CQs are displayed in Table 4.

Table 4: Second Test

Number	Competency Question	Answer	Correct?
1	*What are the activities of Backpackers Destination?*	Adventure, Sports	Yes
2	*Which are the accommodations of Backpackers Destination?*	BudgetAccommodation	Yes
3	*Which are the individuals of the Beach class?*	BondiBeach,CurrawonBeach	Yes
4	*Which are the types of Adventure?*	BunjeeJumping, Safari	Yes
5	*Which are the types of Relaxation?*	Yoga, Sunbathing	Yes
6	*Which are the activities of National Park?*	Hiking	Yes
7	*Is Farmland a rural area?*	Yes	Yes
8	*How many accommodations does a Family Destination have?*	1	Yes
9	*How many activities does a Family Destination have?*	2	Yes
10	*Which are the accommodations of the City?*	LuxuryHotel	Yes

Both CQ number 1 and number 2 are about the concept, Backpackers destination. In number 1, we want to check the activities that can be done at a backpackers destination, according to the ontology, being the classes Adventure and Sports. And in the CQ number 2, we want to know about the accommodation for this type of destination, which only has the Budget accommodation class.

In CQ number 3, we would like to check the individuals of the *Beach* class.

In CQs number 4 and 5, we want to check who they are subclasses of them.

CQ number 6 checks what the activities of *National park* class are.

In CQ 7 we would like to check if the *farmland* is indeed a *rural area*, according to the ontology.

CQs 8 and 9 are about the family destination class. In number 8, we want to check how many accommodations it can have. On the other hand, in CQ number 9, we want to know how many activities there can be. Finally, in CQ 10, we want to check what accommodation a city has.

4.3 Third test: Pizza ontology

And for the last round, we used the Pizza ontology to validate the applicability of the created CQs, considering the content required for representing this ontology. We used the CQs displayed in Table 5.

Table 5: Third test

Competency Question	Answer	Correct?

What is the spiciness of a chicken Topping?	Mild	Yes
Which are the Toppings of a Vegetarian Pizza?	NOT (MeatTopping and FishTopping)	Yes
Which are the Toppings of the Cheesey Pizza?	CheeseTopping	Yes
What is the base of Real Italian Pizza?	ThinAndCrispyBase	Yes
Are Chicken Toppings disjoint from Ham Toppings?	Yes	Yes
What is the country of Origin of Mozzarella Topping?	Italy	Yes
Which are the types of Cheese Topping?	CheeseyVegetableTopping, MozzarellaTopping, GorgonzolaTopping GoatsCheeseTopping, ParmesanTopping FourCheesesTopping	Yes
How many toppings does the Interesting Pizza have?	3	Yes
Which are the instances of the Country?	France,England, Italy,Germany,America	Yes
Which are the classes disjoint of Vegetarian Pizza?	NonVegetarianPizza	Yes

Figure 7 displays the result of the CQ “What is the base of Real Italian Pizza?”. Note that, despite the fact the class name is not explicit in the CQ, since the name in the ontology is "RealItalianPizza”, the CQChecker was able to verify the CQ.

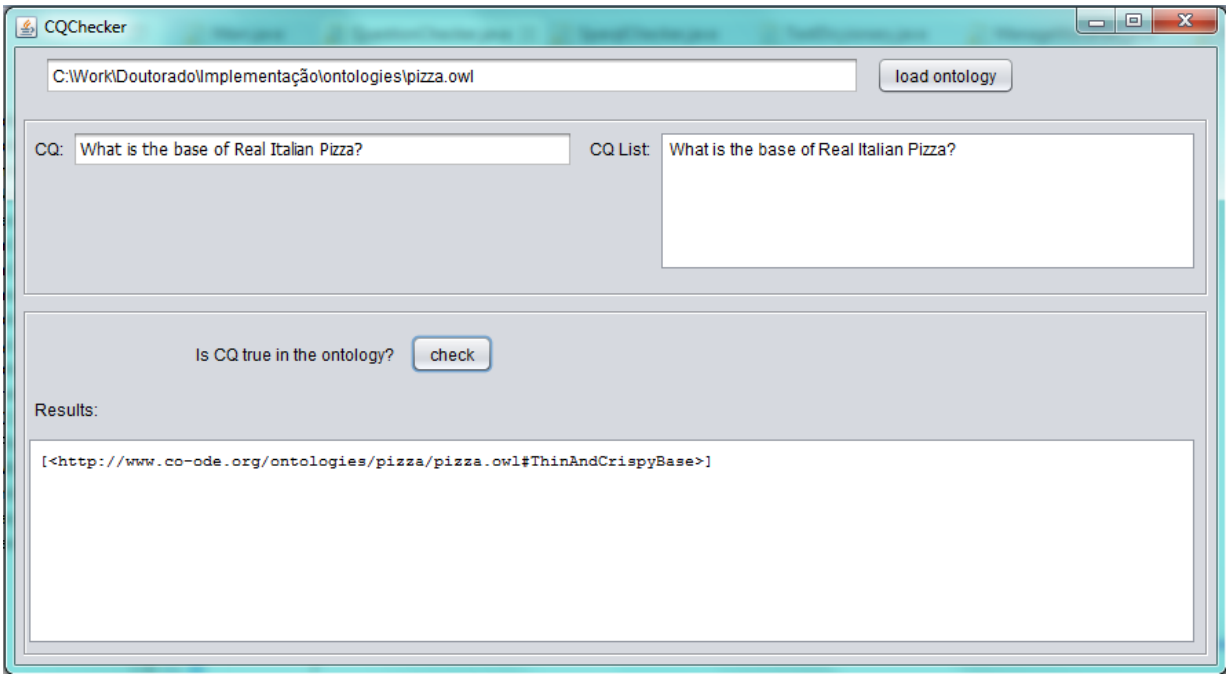

Figure 7: Result of the CQ "What is the base of Real Italian Pizza?"

5 Related Work and Discussion

Few of engineering methodologies cite how, for what purpose, and by which means the ontology engineer has to use CQs. However, there are few proposals of an automated way to create and check ontologies.

The OntoKick tool was distributed with OntoEdit [18]. OntoKick enabled requirement specification creation. In addition, it allows the extraction of relevant structures of a given ontology to build semi-formal ontology descriptions. This approach allows the user to create an ontology from the CQ during the early development stage. In the current work, we propose the

usage of the CQ during the whole ontology development, mainly to validate if the created ontology meets all of the requirements.

Fernandes and colleagues [19] propose the application of Tropos methodology [20] to specify the CQ. Tropos is a Multi-agent system development methodology. The idea behind Tropos is to enable the definition of CQ by using the idea of goal modeling. But there is no automation proposed to check CQs.

Recently, [21] there was a proposal for an approach to deal with ontology authorship by using competency questions and testing - before software development. According to them, ontology authorship is a difficult task for users who are not proficient in Logic. Because of that, they suggest using competency questions written in controlled natural language to resolve this problem.

As shown by the results, our approach is capable of dealing with different types of competency questions. For example, to get the CQChecker answers to the query "***Which are the Toppings of the Veneziana Pizza?***", it is necessary to deal with class union, universal and existential quantifications. Figure 8 portrays the description of the Veneziana Pizza. For this CQ, the CQChecker shows all toppings that meet this description.

However, we were able to identify a few deficiencies in the CQChecker, such as:

- The tool only treats simple English sentences by identifying key words. It is necessary to build a mechanism to accept complex sentences like "Does a bouquet or body of a specific wine change with vintage?".
- The search for the main concept and main property, in the QuestionChecker, is not very efficient yet because we use keywords and similarity for searching.
- In the InferenceChecker no explanation of why the CQ is false has been given yet.

To solve these deficiencies, the current implementation might be improved with other controlled natural language techniques and tools.

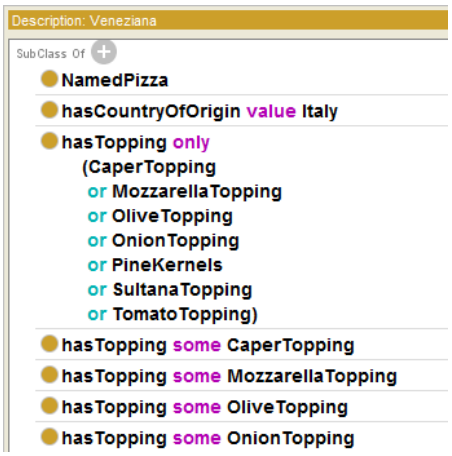

Figure 8: Veneziana Pizza

6 Conclusion

In this paper, we propose a tool, called CQChecker, to enable the creation and checking of CQ with the support of controlled natural language. Furthermore, we provide a mechanism to check whether the ontology answers CQs not only on the assertional level, but also on the terminological level (to some extent).

As described in the literature, CQs perform an important role in the ontology development life cycle, as they represent the ontology requirements. As seen in this work, the traditional methodologies have treated CQs superficially; DL terminological CQs have to be checked manually when they are not already explicitly stated in the ontology.

CQChecker allows the ontology engineer to properly verify CQs automatically (with little user interaction) whether the CQs required for creating the ontology are being fulfilled or not. As CQs are frequently written in controlled natural language, the current approach may help in validating the CQs. Moreover, CQChecker depends little on user interaction during the checking process, mainly just to confirm that they got the correct information from the CQ, which is necessary to obtain the final result. Thus it provides a semi-automatic mechanism, resulting in an innovative feature for the use of DL CQs, as a whole.

CQChecker may be able to reduce the gap seen in the literature between the current status of CQs description and validation. The results obtained so far are encouraging in terms of the variability of OWL queries, which return only a small margin of error, due to the limitations previously discussed, such as not treating CQs composed by compound and complex sentences.

Regarding future work, we intend to overcome the described limitations and assess our tool with more tests, and with larger ontologies and CQs with complex syntactic structure. Hence, we wish to evaluate CQ consistency in order to achieve better results. Moreover, we intend to carry out tests with a representative group of people that do not participate in the development of the tool.

7 References

[1] H. F. Hofmann and F. Lehner. "Requirements Engineering As a Success Factor in Software Projects". *IEEE Softw*., vol. 18, no. 4, pp. 58–66, July 2001.

[2] M. Uschold and M. Gruninger. "Ontologies: principles, methods, and applications". *Knowledge Engineering Review*, vol.11, no. 2, pp. 93–155, 1996.

[3] M. Fernández-López, A. Gómez Pérez and N. Juristo. "METHONTOLOGY: From Ontological Art Towards Ontological Engineering". In *Ontological Engineering on Spring Symposium Series*, Stanford, 1997.

[4] M. del Carmen Suárez-Figueroa, A. Gómez-Pérez, E. Motta and A. Gangemi, editors. *Ontology Engineering in a Networked World*. Springer, 2012.

[5] N. F. Noy and C. D. Hafner. "The state of the art in ontology design: A survey and comparative review". *AI Magazine*, vol. 18, pp. 53–74, 1997.

[6] M. S. Fox and M. Gruninger. "Ontologies for Enterprise Integration". In *CoopIS*, pp. 82–89, 1994.

[7] D. L. McGuinness and F. van Harmelen. "OWL Web Ontology Language Overview". Technical Report REC-owl-features-20040210, W3C, 2004.

[8] F. Santana, D. Schober, Z. Medeiros, F. Freitas and S. Schulz. "Ontology Patterns for Tabular Representations of Biomedical Knowledge on Neglected Tropical Diseases". *Bioinformatics*, vol. 27, no. 13, pp. i349–i356, July 2011.

[9] W3C. "SPARQL Query Language for RDF". Available at http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/. Accessed in October, 2014.

[10] G. Guizzardi and T. Halpin. "Ontological Foundations for Conceptual Modelling". *Appl. Ontol.*, vol. 3, no. 1-2, pp. 1–12, January 2008.

[11] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi and P. F. Patel-Schneider, editors. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, New York, NY, USA, 2003.

[12] T. R. Gruber. "A translation approach to portable ontology specifications". *Knowledge Acquisition*, vol. 5, no. 2, pp.199–220, June 1993.

[13] F. Freitas. "A Connection Method for Inferencing over the Description Logic ALC". In *Description Logics*, edited by R. Rosati, S. Rudolph and M. Zakharyaschev, volume 745. CEUR-WS.org, 2011.

[14] M. Horridge and S. Bechhofer. "The OWL API: A Java API for OWL ontologies". *Semantic Web*, vol. 2, no. 1, pp. 11–21,2011.

[15] Protégé. "Protégé Ontology Library". Available in http://protegewiki.stanford.edu/wiki/Protege_Ontology_Library. Accessed in October, 2014.

[16] G. A. Miller. "WordNet: A Lexical Database for English". *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, November 1995.

[17] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur and Y. Katz. "Pellet: A practical OWL-DL reasoner". *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, no. 2, pp. 51–53, June 2007.

[18] Y. Sure, M. Erdmann, J. Angele, S. Staab, R. Studer and D. Wenke. "OntoEdit: Collaborative Ontology Development for the Semantic Web". In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, ISWC '02, pp. 221–235, London, UK, UK, 2002. Springer-Verlag.

[19] P. C. B. Fernandes, R. S. S. Guizzardi and G. Guizzardi. "Using Goal Modeling to Capture Competency Questions in Ontology-based Systems". *JIDM*, vol. 2, no. 3, pp. 527–540, 2011.

[20] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia and J. Mylopoulos. "Tropos: An Agent-Oriented Software Development Methodology". Autonomous Agents and Multi-Agent Systems, vol. 8, no. 3, pp. 203–236, May 2004.evelopmentMethodology". *Autonomous Agents and Multi-Agent Systems*, vol. 8, no. 3, pp. 203–236, May 2004.

[21] Y. Ren, A. Parvizi, C. Mellish, J. Z. Pan, K. van Deemter and R. Stevens. "Towards Competency Question-driven Ontology Authoring". In *Proc. of 11th Conference on Extended Semantic Web Conference*