Towards Competency Question-Driven Ontology Authoring

Yuan Ren[1], Artemis Parvizi[1], Chris Mellish[1], Jeff Z. Pan[1],
Kees van Deemter[1], and Robert Stevens[2]

[1] Department of Computing Science, University of Aberdeen, Aberdeen, UK
[2] School of Computer Science, University of Manchester, Manchester, UK

Abstract. Ontology authoring is a non-trivial task for authors who are not proficient in logic. It is difficult to either specify the requirements for an ontology, or test their satisfaction. In this paper, we propose a novel approach to address this problem by leveraging the ideas of competency questions and test-before software development. We first analyse real-world competency questions collected from two different domains. Analysis shows that many of them can be categorised into patterns that differ along a set of features. Then we employ the linguistic notion of presupposition to describe the ontology requirements implied by competency questions, and show that these requirements can be tested automatically.

Keywords: #eswc2014Ren.

1 Introduction

In recent years, ontologies based on Description Logics [1] have been widely accepted as an important means for representing and formalising knowledge in different applications [15]. For example, the SNOMED CT (Systematized Nomenclature of Medicine-Clinical Terms) [19] ontology has been mandated for use in over thirty countries.

Ontology authoring remains a challenging task. Studies on ontology authoring such as experiences from the OWL Pizzas tutorial [18] and the NeOn project [7] suggest that ontology formalisms are often not straight-forwardly comprehensible and logical implications can be difficult to resolve. This is because ontology authors are usually domain experts but not necessarily proficient in logic. While it may be difficult for them to express their requirements for the axiomatisation of an ontology, it is also difficult to know whether the requirements are fulfilled as a result of their ontology authoring actions. Hence, ontology authoring is usually time consuming, error-prone and requires extensive training and experience [18].

As a first step towards *Competency Question-driven Ontology Authoring* (CQOA), we address the problems outlined by leveraging the ideas of *competency questions* and *testing driven software development* (where a suite of tests represents a specification for a programme and the tests are coded against). A competency question (CQ) [23] is a natural language sentence that expresses a pattern for a type of questions people expect an ontology to answer. The *answerability* of CQs hence becomes a functional requirement of the ontology. For example, in a software engineering ontology, in order to

V. Presutti et al. (Eds.): ESWC 2014, LNCS 8465, pp. 752–767, 2014.

support the answering of the question "Which process implements a given algorithm?", the ontology should contain concepts *Process* and *Algorithm*, and their instances should be able to have a relation called *Implements*. Also the ontology should (in ways we will investigate below) make it meaningful to ask the question "Which process implements algorithm X?" for any algorithm "X" in the ontology. To investigate such characteristics of ontologies, we are more interested in checking if a CQ can be meaningfully answered, instead of directly answering a CQ. Hence, our research questions are:

1. How are real-world CQs formulated?
2. How can we automatically test whether a CQ can be meaningfully answered?

In this paper, we answer question 1 through the study of real-world CQs in different domains and composed by real ontology authors/users with different levels of expertise. We categorise CQs into several frequent patterns that differ along a set of features and show that CQs collected by us and investigated in previous work can be covered by such a framework. To answer question 2, we employ the notion of presupposition from linguistics to capture the ontology requirements implied by CQs. We show that these presuppositions can be tested automatically at authoring time.

2 Competency Question-Driven Ontology Authoring

2.1 Ontological Artifacts in Description Logics

In general, when specified in a Description Logic (DL), an ontology uses classes, properties and their instances to describe the domain of discourse. Atomic classes such as $CheeseTopping$, $Pizza$ and atomic properties such as $hasTopping$ can be connected with DL constructors to compose complex classes. For example, $Pizza \sqcap \exists hasTopping.CheeseTopping$ means *pizzas that have at least one cheese topping*. The relationships between classes, properties and instances are described with ontology axioms. Considering the following typical axioms:

$$CheeseyPizza \equiv Pizza \sqcap \exists hasTopping.CheeseTopping \quad (1)$$

$$AmericanPizza \sqsubseteq Pizza \quad (2)$$

$$AmericanPizza \sqsubseteq \exists hasTopping.MozzarellaTopping \quad (3)$$

$$MozzarellaTopping \sqsubseteq CheeseTopping \quad (4)$$

Axiom (1) means that *cheesy pizzas are those pizzas that have at least one cheese topping*, in which $\equiv$ denotes an equivalence. Axiom (2) means that *American pizza is a pizza*, in which $\sqsubseteq$ denotes a subsumption relation. Axiom (3) means that *American pizza contains at least one mozzarella topping*. And axiom (4) means that *mozzarella topping is a cheese topping*.

A formal ontology consists of a set of axioms. These axioms describe the explicit knowledge in the ontology and can interact with each other to infer implicit knowledge. For example, by combining axioms (3) and (4) we can infer that $AmericanPizza \sqsubseteq \exists hasTopping.CheeseTopping$. Combining with axioms (1) and (2) we can further infer that $AmericanPizza \sqsubseteq CheeseyPizza$, i.e. *American pizza is a cheesey pizza*. Such inference can be realised by an automatic reasoner.

When a class expression C can be instantiated (i.e. it is possible for it to have instances in the domain), we say that C is *satisfiable*. Checking whether a class is satisfiable in an ontology is a reasoning task that can also be accomplished by a reasoner.

2.2 Presupposition and Cooperative Question Answering

Following on from ideas of Frege, many philosophers of language use the term *presupposition* to refer to a special condition that must be met for a linguistic expression to have a denotation [2]. For example, the question "have you stopped feeding your dog?" presupposes that the addressee has a dog and has been feeding it; it can only be successfully answered if these conditions are satisfied – otherwise the question is in some sense meaningless.

The fact that a question may have presuppositions, and that these may represent misconceptions on the part of the asker, has been exploited by researchers working on principles for cooperative question-answering from databases [9]. For instance, if a user asks "Who passed CMSC 420 in the fall semester of 1991?", the answer *nobody* is not cooperative if in fact CMSC 420 was not taught in the fall semester of 1991. In this case, the user should be alerted to the failed presupposition in their question. Such a failure can be computed in this case by detecting a subpart of the original question that produces an empty set of results (informally, *CMSC 420 in the fall semester*) [10].

In this paper we take a similar approach, identifying presuppositions of *competency questions* associated with an ontology. In this case, the aim is not to provide better answers to the questions but to help the user detect when their ontology is out of step with the kinds of questions they would like to be able to ask.

2.3 From Competency Questions To Authoring Tests

(Informal) CQs are expressions of questions that an ontology must be able to answer [23]. We consider these to be natural language sentences that express patterns for types of question people want to be able to answer with the ontology. The ability to answer questions of the type indicated by a CQ meaningfully can be regarded as a *functional requirement* that must be satisfied by the ontology.

Example 1. Below are some example CQs:

- "Which mammals eat grass?" (in an animal ontology)
- "Which processes implement an algorithm?" (in a software engineering ontology)

The first of these suggests a specific pattern, which (when the ontology is complete) could perhaps be expressed as a single SPARQL[1] query such as:

```
select ?m where
  {?m type Mammal . ?g type Grass . ?m eat ?g}
```

[1] http://www.w3.org/TR/sparql11-overview/

The second (interpreted with "an" having wide scope[2]) provides a pattern that will apply to a number of possible queries. This pattern might be thought of as a template for SPARQL queries, where certain slots will be filled in before the query is presented to the ontology. When the ontology is complete, this might look something like:

```
select ?p where
  {?p type Process . $X type Algorithm . ?p implements $X}
```

where `$X` is to be filled in (at query presentation time) with whatever algorithm (in the ontology) in which the user is interested.

The examples show that a CQ can suggest a single desired query (as in the first case) or a set of possible queries (as in the second). In the following, we will abstract away from this distinction and, for instance, talk about "answering a CQ" as a shorthand for "answering the queries implied by a CQ".[3]

Compared to more formal requirement specifications, CQs are particularly useful to ontology authors less familiar with DLs because CQs are in natural language, are about domain knowledge, and do not require understanding of DLs. Hence in ontology authoring practice, CQs help authors to determine the scope and granularity of the ontology, and to identify the most important classes, properties and their relations.

From a linguistic point of view, such questions also have presuppositions about the domain of discourse that have to be satisfied:

Example 2. In order to meaningfully answer the CQ "Which processes implement an algorithm?" it is necessary for the ontology to satisfy the following presuppositions:

1. Classes $Process$, $Algorithm$ and property $implements$ occur in the ontology;
2. The ontology allows the possibility of $Process$es implementing *Algorithm*s;
3. The ontology allows the possibility of $Process$es *not* implementing $Algorithm$s.

The last two of these perhaps need some justification. If case 2 were not satisfied, the answer to all the queries (for all `$X`) would be "none", because the ontology could never have a $Process$ implementing an $Algorithm$. This would be exactly the kind of uncooperative answer looked at by the previous work on cooperative question-answering[9]. It is hard to imagine an ontology author really wanting to retrieve this information. Rather, this can be taken as evidence of possible design problems in the ontology. If case 3 were not satisfied, the answer to all the queries (for all `$X`) would be a list of all the $Process$es. This would mean that the questions would be similarly uninteresting to the ontology author, again signalling a possible problem in the ontology.

CQs can have clear and relatively simple syntactic patterns. For example, the CQs in Example 1 are all of the following semi-formal pattern:

$$Which\ [CE1]\ [OPE]\ [CE2]?$$

[2] Alternative formulations with the same intention might be "Which processes implement a given algorithm?", "For any algorithm, what processes implement it?" or "Which processes implement this algorithm?".

[3] It is also possible to formulate the queries in a way such that the answers to the CQ are not instances of $Mammal$ or $Process$, but their sub-classes. Nevertheless, our discoveries presented later in the paper will not be affected by such a difference.

where $CE1$ and $CE2$ are class expressions (or individual expressions as a special case) and OPE is an object property expression. This pattern asks for instances or subclasses of $CE1$ that can have an OPE relation to some instance of $CE2$. With such patterns, the presuppositions shown in Example 2 can be verified automatically:

1. $CE1$, $CE2$ and OPE should occur in the ontology;
2. $CE1 \sqcap \exists OPE.CE2$ should be satisfiable in the ontology;
3. $CE1 \sqcap \neg(\exists OPE.CE2)$ should be satisfiable in the ontology. Here $\neg$ is the constructor for negation.

We call tests of this kind which can be derived from CQs *Authoring Tests* (ATs).

The idea of CQOA is to support the ontology author in the formulation of machine processable CQs for their ontology. In an implemented system, users will be allowed to either import their predefined CQs or enter new CQs in a controlled natural language. The authoring environment will identify the patterns of the inputted CQs and generate appropriate ATs. With the ATs, certain aspects of the answerability of the CQs can then be tested by the authoring environment to find places where the ontology does not yet meet the requirements. If there is a change in the status of these ATs from true to false or vice versa, the system will report the result to the users. The pattern identification, AT generation and testing procedures are all transparent to authors hence they can be utilised by novice ontology authors.

3 Related Work

Exploring competency questions (CQs) in ontology development is not a new idea in itself [23,17,21]. The NeOn methodology [20] has worked towards an ontology specification task, which results in a set of natural language CQs. A visual solution based on a goal-based methodology for capturing CQs has been presented in [8]. A formalisation of CQs into SPARQL queries [24] and CQs into DL queries [13] have also been implemented. An algorithm for checking natural language CQs has been developed by [3]. Nevertheless, these works focused on limited forms of CQs such as "What is ...?", "How much ...?". A wider spectrum of CQs and their usefulness in ontology authoring were not investigated. Moreover, they are more concerned with answering particularly CQs, but less with whether the answers are meaningful w.r.t. (with respect to) the presuppositions.

Testing is also widely used in different ontology authoring systems to provide feedback to authors on the quality of the ontology. The Rabbit interface [5] and the Simplified English prototype [16] offer syntactic checking such as incorrect words or disallowed input. Systems such as Protégé[4] and OntoTrack [11] use reasoners to offer basic semantic checking such as inconsistency checking. Systems such as Roo [6] intend to advise the user of the potential authoring consequences. Justification engines [11] are also used to explain why certain deductions have been made. Systems such as the OWL Unit Test Framework in Protégé, Tawny-OWL [12] and OntoStudio[5] allow users

[4] `http://protege.stanford.edu`

[5] `http://www.semafora-systems.com/en/products/ontostudio/`

to define unit tests and run them in the authoring environment. Generic tests such as consistency, input validity do not capture the requirements that are specific to the ontologies in question. The author-defined tests allow the expression of such requirements but require further knowledge and skills of ontology technologies. For novice authors, designing a test suite for an ontology is hardly easier than designing the ontology itself.

4 An Empirical Study of Competency Questions

In contrast to the existing work, we combine CQs and testing in ontology authoring, using CQs as a means for novice authors to express requirements, and derive tests from these CQs to capture their presuppositions. We aim to understand the different kinds of CQs that are asked by authors in real-world scenarios, to ensure that the ontology can respond optimally to them. We therefore address research question 1 by analysing real-world CQs.

4.1 Competency Question Collection

Due to the flexibility permitted in CQ construction, it was not feasible for us to enumerate all possible CQs. In order to cover CQs used in different domains and from authors with different levels of expertise, we collected 92 CQs from the Software Ontology Project[6] and 76 CQs from the Manchester OWL Tutorials in 2013. The software ontology project seeks to describe software such that software registries and repositories can be adequately tagged and indexed; it is also used to describe the software that used in the analysis of data. CQs in this project are proposed by the users of this ontology and hence represent requirements from a professional point of view. The OWL tutorials were events where basic ontology technologies were taught to participants, who were mostly novice authors. In the tutorials , the Pizza ontology[7] was used as a show case ontology and participants were asked to write CQs they would like to get answered with the pizza ontology working as part of an 'intelligent pizza finder' application.

After obtaining the collection of questions, we removed invalid CQs, including:

1. Redundant questions;
2. Incomplete sentences that cannot be properly understood. For example, in the software collection, one question is "What level of expertise is required?". In this question, it is not clear what the expertise is required for.
3. Sentences that are not really CQs. For example, in the pizza collection there is one question "Should we include the oven type in the pizza definition? (eg wood fired vs electric oven)". Nevertheless, "What oven type is this pizza?" can be regarded as a valid CQ.
4. Questions beyond the expressive power of a DL-based ontology language. For example, in the software collection a question asks "How can I get problems fixed?".

[6] http://softwareontology.wordpress.com/2011/04/01/user-sourced-competency-questions-for-software/

[7] http://130.88.198.11/co-ode-files/ontologies/pizza.owl

The answer to such a *How* question should be a procedure that involves conditions and actions. Whilst an ontology is mainly used for modelling of static domain knowledge instead of dynamic procedures.

With the above invalid CQs removed, we obtained 75 valid CQs in the software collection and 70 in the pizza collection.

4.2 A Framework for Patterns of Competency Questions

We analysed the collected CQs to identify the patterns to which they belong. We are more interested in the semantic meaning of the CQs than their surface form. Hence we omit the syntactic differences between variations with the same semantic meaning.

In order to represent the commonality and variability of different CQ patterns, we employed the feature-based modelling method [14] and describe different CQ patterns w.r.t. a set of features identified from our CQ collections:

1. **Question Type** determines the kinds of answer presented when answering the CQ:
   (a) *Selection question* should be answered with a set of entities or values that satisfy certain constraints. The CQs in Example 1 are all selection questions.
   (b) *Binary question* should be answered with a boolean value, i.e. *yes* or *no*, indicating the existence of any answer to a selection. For example, "Does this pizza contain halal meat?" is a binary question corresponding to a selection question "Which of these pizzas contain halal meat?".
   (c) *Counting question* should be answered with the number of different answers to a selection question. For example, "How many pizzas have either ham or chicken topping?" is a counting question. Its corresponding selection question is "Which pizzas have either ham or chicken topping?".
2. **Element Visibility** indicates whether the modelling elements, such as the class expressions and property expressions are *explicit* or *implicit* in the CQ. For example, "What are the export options for this software?" has explicit elements *Software* and *Export Option*, but also an implicit relation *hasExportOption* between softwares and export options. Note that even implicit elements should occur in the ontology to make the CQ meaningful.
3. **Question Polarity** determines if the question is asked in a *positive* or *negative* manner, e.g. "Which pizzas contain pork?" *v.s.* "Which pizza has no vegetables?".
4. **Predicate Arity** indicates the number of arguments of the main predicate:
   (a) *Unary predicate* is concerned with a single set of entities/values and its instances, e.g. "Is it thin or thick bread?".
   (b) *Binary predicate* is concerned with the relation between 2 sets of entities/values and their instances, such as the eat and $implement$ in Example 1.
   (c) *N-ary predicate* is concerned with the relation among multiple ($\geq$ 3) sets of entities/values and their instances. Given the fact that DLs can only represent unary and binary predicates, an N-ary predicate has to be represented as a concept via reification. In the next section we will show how this affects the ATs.
5. **Relation Type** indicates the kind of relation for the main relation involved in the CQ. As in DLs, CQs can have object property relations or datatype property relations. Note that a relation with more than 2 arguments or with its attributes has to be represented by an entity via reification.

6. **Modifier** is employed to impose restrictions on some entities/values:
   (a) *Quantity modifier* restricts the number of relations among entities/values.
      i. It can be a concrete value or value range. For example "If I have 3 ingredients, how many kinds of pizza I would make?" has a quantity modifier 3 on the number of pizza-ingredient relations for each pizza.
      ii. It can be a superlative value or value range. For example, "Which pizza has the most toppings?" has a quantity modifier *most* on the number of pizza-topping relations for each pizza.
      iii. It can also be a comparative value or value range. For example, "Which pizza has more meat than vegetables?" has a quantity modifier *more* on the number of pizza-meat and pizza-vegetable relations for each pizza.
   (b) *Numeric modifier* is used to restrict the value of some datatype properties. Similarly to the quantity modifier, it can be a concrete value or value/range, or a superlative value, or a comparative value. For example, "What pizza has very little ($\leq 10\%$) onion and/or leeks and/or green peppers?"
7. **Domain-independent Element** is an element that can occur across different knowledge domains. It is usually associated with some physical or cognitive measurements. Some most commonly used domain-independent elements include:
   (a) *Temporal element* in the CQ indicates that the CQ is about the time of some event, e.g. "When was the 1.0 version released?".
   (b) *Spatial element* in the CQ indicates that the CQ is about the location of some event. It does not have to be a physical location. For example "Where is the documentation?" can be answered with a file path or a URL.

We consider the *Question Type*, *Element Visibility* and *Question Polarity* as secondary features as their variabilities do not change the required modelling elements of the ontology. All other features are primary features. CQs with different primary features are distinguished into different archetypes. CQs with different secondary features in an archetype are distinguished into different sub-types. Together, they constitute a generic framework to formulate different CQ patterns. For example, the CQ pattern *Which [CE1] [OPE] [CE2]*? features a selection question with binary predicate of an object property relation and all elements are explicit.

4.3 Result of the Study

With the feature-based framework, we identify 12 archetypes of CQ patterns in our collection. They are shown in Table 1. The 1st column shows the ID of the archetype, the 2nd and 3rd columns show the pattern and 1 example from our collection. The last 4 columns are the primary features. As we mentioned above, some archetype patterns have sub-types. An example of the sub-types of archetype 1 is illustrated with Table 2, in which the last 3 columns are the secondary features.

The archetypes and sub-types of CQ patterns we have identified cover all the CQs in our collection, but we do not know directly how many CQs for other domains they will cover. Nevertheless, the feature-based framework is flexible enough to describe CQs we have not encountered. For example, a hypothetical CQ "How many pieces of software are most efficient when providing this service?" has a pattern *How many [CE1] are [NM] to [OPE] [CE2]*?, which is a counting question sub-type in archetype 6.

Table 1. CQ Archetypes (PA = Predicate Arity, RT = Relation Type, M = Modifier, DE = Domain-independent Element; obj. = object property relation, data. = datatype property relation, num. = numeric modifier, quan. = quantitative modifier, tem. = temporal element, spa. = spatial element; CE = class expression, OPE = object property expression, DP = datatype property, I = individual, NM = numeric modifier, PE = property expression, QM = quantity modifier)

ID	Pattern	Example	PA	RT	M	DE
1	Which [CE1] [OPE] [CE2]?	Which pizzas contain pork?	2	obj.		
2	How much does [CE] [DP]?	How much does Margherita Pizza weigh?	2	data.		
3	What type of [CE] is [I]?	What type of software (API, Desktop application etc.) is it?	1			
4	Is the [CE1] [CE2]?	Is the software open source development?	2			
5	What [CE] has the [NM] [DP]?	What pizza has the lowest price?	2	data.	num.	
6	What is the [NM] [CE1] to [OPE] [CE2]?	What is the best/fastest/most robust software to read/edit this data?	3	both	num.	
7	Where do I [OPE] [CE]?	Where do I get updates?	2	obj.		spa.
8	Which are [CE]?	Which are gluten free bases?	1			
9	When did/was [CE] [PE]?	When was the 1.0 version released?	2	data.		tem.
10	What [CE1] do I need to [OPE] [CE2]?	What hardware do I need to run this software?	3	obj.		
11	Which [CE1] [OPE] [QM] [CE2]?	Which pizza has the most toppings?	2	obj.	quan.	
12	Do [CE1] have [QM] values of [DP]?	Do pizzas have different values of size?	2	data.	quan.	

Table 2. CQ Sub-types of Archetype 1 (QT = Question Type, V = Visibility, QP = Question Polarity, sel. = selection question, bin. = binary question, cout. = counting question, exp. = explicit, imp. = implicit, sub. = subject, pre. = predicate, pos. = positive, neg. = negative)

ID	Pattern	Example	QT	V	QP
1a	Which [CE1] [OPE] [CE2]?	What software can read a .cel file?	sel.	exp.	pos.
1b	Find [CE1] with [CE2].	Find pizzas with peppers and olives.	sel.	imp. pre.	pos.
1c	How many [CE1] [OPE] [CE2]?	How many pizzas in the menu contains meat?	cout.	exp.	pos.
1d	Does [CE1] [OPE] [CE2]?	Does this fotware provide XML editing	bin.	exp.	pos.
1e	Be there [CE1] with [CE2]?	Are there any pizzas with chocolate?	bin.	imp. pre.	pos.
1f	Who [OPE] [CE]?	Who owns the copyright?	sel.	imp. sub.	pos.
1g	Be there [CE1] [OPE]ing [CE2]?	Are there any active forums discussing its use?	bin.	exp.	pos.
1h	Which [CE1] [OPE] no [CE2]?	Which pizza contains no mushroom?	sel.	exp.	neg.

After obtaining the competency question patterns, we analysed the distribution of each pattern in our two scenarios. The numbers of competency questions belonging to each archetype are shown in Table 3. We can see from this that among the 12 archetypes, 9 can be observed in the software collection, 9 can be observed in the pizza collection,

Table 3. Numbers of CQs in Each Archetype Pattern

Archetype	1	2	3	4	5	6	7	8	9	10	11	12
Software Collection	38	11	1	1	0	4	5	5	3	7	0	0
Pizza Collection	23	7	4	0	5	1	0	22	0	2	5	1
Total	61	18	7	1	5	5	5	27	3	9	5	1

and 6 are shared by both collections. These 6 are also the most populated archetypes, together covering 86.2% of all the collected CQs. This suggests that we might have begun to find a kind of closure in terms of the most significant CQ types and that further domains may not introduce many more important types.

We also examined the applicability of our framework to the 55 CQs mentioned in previous work [20,8,24,13,3]. Most of those CQs are covered by our framework and archetype 1 is the most populated one. The *only* CQ not definitely covered is "Why universities are organised into departments?" [24]. This can be categorised to archetype 2 if the ontology represents the answer to *why* with a textual string. However, we believe a proper modelling of such questions would require more complex formalisation.

5 Answerability of Competency Questions

In this section, we try to address research question 2 by generating ATs from CQs and showing that these ATs can be checked automatically. In contrast to previous work that attempts to find answers to concrete CQs, we investigate whether or not the CQs can be meaningfully answered.

5.1 Presuppositions in Competency Question Features

In CQOA, we are interested in whether the ontology contains the knowledge required to answer CQs meaningfully. Such knowledge requirements are closely related to the presuppositions in the CQs.

Given that our framework describes the CQs in terms of a set of features, we first analyse the presuppositions implied by different variations of each feature:

1. **Question Type:** regardless of the question type, the modelling elements mentioned in the question should **occur** in the ontology. Classes should also be **satisfiable**.
   (a) *Selection question* asks for the answers satisfying certain constraints. The ontology should allow some answers to satisfy the constraints. For example, "Which pizzas contain pork?" implies that pork is allowed to be contained in pizzas, i.e. $Pizza \sqcap \exists contains.Pork$ should be **satisfiable**. Otherwise, no pizza can contain pork at all. The ontology should also allow some entities to NOT satisfy the constraints. For example, the CQ above implies that it is possible for some pizza to contain no pork, i.e. $Pizza \sqcap \forall contains.\neg Pork$ is satisfiable. Otherwise, any pizza must contain pork and the *"contains pork"* part in the CQ becomes useless.

(b) *Binary question* asks whether there is an answer satisfying the constraint. It does not have the two satisfiability presuppositions.
(c) *Counting question* asks for the number of the answers satisfying the constraints. It assumes the possibility of some answer satisfying the constraint and also some answer not satisfying it. Hence it has the satisfiability presuppositions.

2. **Element Visibility:** regardless of the visibility of a modelling element, it should always occur in the ontology to make the CQ answerable. Nevertheless, an implicit element does not appear in the CQ hence its corresponding name in the ontology cannot be directly obtained. This name can be derived from related entities. For example, in "What are the export options for this software?" we can name the implicit relation *hasExportOption*. Otherwise it can be assigned by the author.
3. **Predicate Arity:** the arity of the predicate affects how it should occur in the ontology. Modern ontology languages support both unary (i.e. classes) and binary (i.e. properties) predicates. Hence their names can directly occur in the ontology. However, N-ary predicate has to be represented as a class via reification. This leads to the occurrence of other implicit predicates. For example, in "What is the best software to read this data?" the predicate *read* has 3 arities, namely the *software*, the *data*, and the *performance*. Hence $Reading$ should occur in the ontology as a *Class* instead of a *Property*. Moreover, there should be 3 more implicit predicates, namely the $hasSoftware$, the $hasData$ and the $hasPerformance$.
4. **Relation Type:** as the name suggests, the meta-type of a property occurring in the ontology is determined by the type of relation it represents in the CQ. In other word, if a property P is between two entities, then it is presupposed that P is an **instance** of `OWL:ObjectProperty`. If P is between an entity and a value, then it is presupposed that P is an instance of `OWL:DatatypeProperty`.
5. **Modifier:** the modifiers further impose restrictions on answers of the CQ.
   (a) **Quantity modifier** has a similar effect as *question type* on the satisfiability presupposition of certain class expressions in the ontology.
      i. If the modifier is a concrete value or range, then as for a *selection question*s it presupposes that potnetial answers are allowed to satisfy, as well as not to satisfy, this modifier. For example, "If I have 3 ingredients, how many kinds of pizza can I make?" implies that the ontology allows pizzas with 3 ingredients and ones with fewer or more than 3 ingredients, i.e. $Pizza \sqcap = 3\ hasIngredient.Ingredient$ and $Pizza \sqcap \neg(= 3\ hasIngredient.Ingredient)$ should both be satisfiable in the ontology.
      ii. If the modifier is a superlative value or value range, then the ontology should allow answers with **multiple cardinality** values on the predicate on which the modifier is imposed. For example, in "Which pizza has the most toppings?" the presupposition is that *pizzas are allowed to have different numbers of toppings* otherwise all pizzas will have exactly the same number of toppings. More formally, this means that for each number $n \geq 0$, $Pizza \sqcap \neg(= n\ hasTopping.Topping)$ should be satisfiable.
      iii. If the modifier is a comparative value or value range, then the ontology should allow an answer with the required **comparative cardinality** values on the different relations being compared, as well answers without the required comparative cardinality values. For example, "Which pizza

has more meat than vegetables?" presupposes that *pizzas are allowed to have more meat than vegetables* otherwise none of the pizza is an answer. More formally this means that for some number $n \geq 0$, $Pizza \sqcap \leq n\ has.Vegetable$ and $Pizza \sqcap \geq n+1\ has.Meat$ should both be satisfiable. It also presupposes that *pizzas are allowed to have no more meat than vegetables* otherwise all pizzas have more meat than vegetable. More formally this means that for some number $n \geq 0$, $Pizza \sqcap \leq n\ has.Meat$ and $Pizza \sqcap \geq n\ has.Vegetable$ should both be satisfiable.

(b) *Numeric modifier* has similar presuppositions to a quantity modifier. In the concrete value or value range case and comparative value case, the CQ carries the presuppositions that the ontology should allow answers satisfying the modifier and those not satisfying the modifier. In the superlative value case, the CQ carries the presupposition that the ontology should allow multiple values on the relation on which the modifier is imposed.
Furthermore, the **range** of the property on which the modifier is imposed must be a comparable datatype, such as *integer*, or *float*, otherwise the question can not be answered meaningfully.

6. **Domain-independent Element** in the CQ can also affect the meta-type and type of some modelling elements in the ontology. The temporal element is usually associated to some temporal datatypes. For example, "When was the 1.0 version released?" has presuppositions that the $wasReleasedOn$ is a datatype property, and that the range of $wasReleasedOn$ is one of the temporal datatype, such as *datatime*. It is possible to use some other datatypes, such as *integer* to denote the year of release, but this is not considered a best practice.
The *spatial element* is not necessarily representing a geographical location hence it is hard to determine the type of its corresponding element in the ontology.

5.2 Formalising the Authoring Tests

From the analysis in Sec. 5.1, we realise that the features in the CQs are related to certain categories of presuppositions. Each of these categories contains parameter(s) derived from the CQ and can be realised by some checking in the ontology. ATs formalise this idea. We summarise the ATs in Table 4. In this table the 1st column are the ATs, the 2nd column are the parameters for each AT and the 3rd column shows how each AT can be checked with ontology technologies. We omit the formalisation of some ATs, such as those associated with comparative numeric modifiers, because such features were not observed in our collection; they can be formalised in a similar manner as the ones in the table.

As one can see, all of these ATs can be checked automatically. *Occurrence* can be checked directly against the ontology. *Meta-Instance* can be checked via RDF reasoning. All the others can be checked with ontology reasoning.

In an implemented system, we offer users a controlled natural language to input CQs based on the patterns identified earlier. Hence the archetype and/or sub-type of input CQs are implicitly specified by users and automatically identified by the system: For example, CQ "What is the best software to read this data?" belongs to archetype CQ pattern 7 $What\ [CE1]\ is\ [NM]\ to\ [OPE]\ [CE2]?$

Table 4. Authoring Tests ($\sqcap$ means conjunction, $\neg$ means negation, $\exists P.E$ means having P relation to some E, $= nP.E$ ($\geq nP.E, \leq nP.E$) means having P relation(s) to exactly (at least, at most) n E(s), $\forall P.E$ means having P relation (if any) to only E, $\top$ means everything)

AT	Parameter	Checking
Occurrence	[E]	E in ontology vocabulary
Class Satisfiability	[CE]	CE is satisfiable
Relation Satisfiability	[CE1]	$CE1 \sqcap \exists P.E2$ is satisfiable, $CE1 \sqcap \neg\exists P.E2$ is satisfiable
	[P]	
	[E2]	
Meta-Instance	[E1]	$E1$ has `type` $E2$
	[E2]	
Cardinality Satisfiability	[CE1]	$CE1\sqcap = nP.E2$ is satisfiable, $CE1\sqcap\neg = nP.E2$ is satisfiable
	[n]	
	[P]	
	[E2]	
Multiple Cardinality (on superlative quantity modifier)	[CE1]	$\forall n \geq 0$, $CE1 \sqcap \neg = nP.E2$ is satisfiable
	[P]	
	[E2]	
Comparative Cardinality (on quantity modifier)	[CE1]	$\exists n \geq 0$, $CE1\sqcap \leq n\ P1.E1$ and $CE1\sqcap \geq n{+}1\ P2.E2$ are satisfiable, $\exists m \geq 0$, $CE1\sqcap \leq m\ P2.E2$ and $CE2\sqcap \geq (m+1)\ P1.E1$ are satisfiable
	[P1]	
	[P2]	
	[E1]	
	[E2]	
Multiple Value (on superlative numeric modifier)	[CE1]	$\forall D \subseteq range(P)$, $CE1 \sqcap \neg\exists P.D$ is satisfiable
	[P]	
Range	[P]	$\top \sqsubseteq \forall P.E$
	[E]	

From the CQ and its pattern the system can automatically extract the features and elements of the CQ: it is a selection question ("What") containing a 3-ary (among "software", "data" and some performance) predicate ("read") with a superlative numeric modifier ("best"), which should be modelled as a class and some implicit object and datatype properties, whose names can be generated from contexts or assigned by users.

Then the system can automatically generate and parameterise the following ATs:

1. Occurrence tests of $Software$, $Data$, $Read$, $hasSoftware$, $hasPerformance$ and $hasData$. The first 3 should occur as classes and the last 3 as properties. $Read$ is the class representation of the "reading" predicate in the CQ;
2. Relation Satisfiability tests of ($Read$, $hasSoftware$, $Software$), ($Read$, $hasData$, $Data$) and ($Read$, $hasPerformance$, $\top$), which guarantee that the ontology allow some $Read$ to be associated with $Software$, $Data$ and to have performance;

3. Meta-Instance test of $(hasSoftware, \texttt{ObjectProperty})$, $(hasData, \texttt{ObjectProperty})$ and $(hasPerformance, \texttt{DatatypeProperty})$, which further specify the meta-types of the 3 properties;
4. Multiple Value on superlative numeric modifier test of $(Read, hasPerformance)$, which guarantees that instances of $Read$ can have *different* performance values;
5. Range test of $(hasPerformance, decimal \cup float \cup double)$, which ensures that the value of $hasPerformance$ must be a comparable numeric value, so that one can find the best performance;

As the pipeline shows, the procedure from CQs (in a controlled natural language) to ATs can be automated. Eventually, all these ATs can be automatically checked and results can be provided to users.

6 Conclusion and Future Work

We have investigated the problem of requirement description and testing in ontology authoring for novice authors. We proposed Competency Question-driven Ontology Authoring (CQOA) by leveraging the ideas of CQs and test before styles of software development. To formally describe different real-world CQs, we have collected CQs from the software and pizza domains and analysed their commonalities and varieties with a set of features. It showed that the CQ patterns we identified covered all the collected CQs. To automatically test whether a CQ can be meaningfully answered, we investigated the presuppositions implied by CQ features. All these presuppositions can be parameterised and formalised into automatic ATs. Although our research were based on CQs in English, our results are transferable to other languages.

In future, we will implement the presented pipeline both as a Protégé plug-in and as a standalone system, supported by the TrOWL reasoner [22]. We will design and conduct experiments with human participation to compare their efficiency and productivity with and without the AT assistance. We are interested in extending the current framework by investigating more CQs, features and presuppositions. For example, it is difficult to formalise the spatial element presupposition in the current framework; we plan to address it by looking into ontology design patterns or foundational ontologies [4]. We would also like to investigate presuppositions of a finer linguistic "granularity". E.g. "Which pizzas contain pork?" appears to presuppose the existence of multiple types of pizza that contain pork while "Which pizza contains pork?" does not. Finally, some ATs such as the cardinality ones (Table 4) lead to a large number of tests, and we plan to investigate optimising the testing of such ATs.

Acknowledgments. This research has been funded by the EPSRC *WhatIf* project (EP/J014176/1) and the EU IAPP K-Drive project (286348). We also thank Caroline Jay and Markel Vigo for their inspiring discussion on CQs in ontology authoring and their assistance in collecting the pizza CQs.

References

1. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
2. Beaver, D.: Presupposition. In: van Benthem, J., ter Meulen, A. (eds.) The Handbook of Logic and Language, pp. 939–1008. Elsevier (1997)
3. Bezerra, C., Freitas, F., Santana, F.: Evaluating ontologies with competency questions. In: 2013 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT), vol. 3, pp. 284–285. IEEE (2013)
4. Borgo, S., Masolo, C.: Foundational choices in dolce. In: Handbook on ontologies, pp. 361–381. Springer (2009)
5. Denaux, R., Dimitrova, V., Cohn, A.G., Dolbear, C., Hart, G.: Rabbit to OWL: Ontology authoring with a CNL-based tool. In: Fuchs, N.E. (ed.) CNL 2009. LNCS, vol. 5972, pp. 246–264. Springer, Heidelberg (2010)
6. Denaux, R., Thakker, D., Dimitrova, V., Cohn, A.G.: Interactive semantic feedback for intuitive ontology authoring. In: FOIS, pp. 160–173 (2012)
7. Dzbor, M., Motta, E., Gomez, J.M., Buil, C., Dellschaft, K., Görlitz, O., Lewen, H.: D4.1.1 analysis of user needs, behaviours & requirements wrt user interfaces for ontology engineering. Technical report (August. 2006)
8. Fernandes, P.C.B., Guizzardi, R.S., Guizzardi, G.: Using goal modeling to capture competency questions in ontology-based systems. Journal of Information and Data Management 2(3), 527 (2011)
9. Gaasterland, T., Godfrey, P., Minker, J.: An Overview of Cooperative Answering. Journal of Intelligent Information Systems 1(2), 123–157 (1992)
10. Kaplan, S.J.: Cooperative Responses from a Portable Natural Language Query System. Artificial Intelligence 19(2), 165–187 (1982)
11. Liebig, T., Noppens, O.: Ontotrack: A semantic approach for ontology authoring. Web Semantics: Science, Services and Agents on the World Wide Web 3(2), 116–131 (2005)
12. Lord, P.: The semantic web takes wing: Programming ontologies with tawny-owl. In: OWLED 2013 (2013)
13. Malheiros, Y., Freitas, F.: A method to develop description logic ontologies iteratively based on competency questions: an implementation. In: ONTOBRAS, pp. 142–153 (2013)
14. Palmer, S.R., Felsing, M.: A practical guide to feature-driven development. Pearson Education (2001)
15. Pan, J.Z., Thomas, E., Ren, Y., Taylor, S.: Tractable Fuzzy and Crisp Reasoning in Ontology Applications. In: IEEE Computational Intelligence Magazine (2012)
16. Power, R.: OWL simplified english: A finite-state language for ontology editing. In: Kuhn, T., Fuchs, N.E. (eds.) CNL 2012. LNCS, vol. 7427, pp. 44–60. Springer, Heidelberg (2012)
17. Presutti, V., Blomqvist, E., Daga, E., Gangemi, A.: Pattern-based ontology design. In: Ontology Engineering in a Networked World, pp. 35–64. Springer (2012)
18. Rector, A., Drummond, N., Horridge, M., Rogers, J., Knublauch, H., Stevens, R., Wang, H., Wroe, C.: Owl pizzas: Practical experience of teaching owl-dl: Common errors & common patterns. In: Engineering Knowledge in the Age of the Semantic Web, Springer (2004)
19. Stearns, M.Q., Price, C., Spackman, K.A., Wang, A.Y.: SNOMED Clinical Terms: Overview of the Development Process and project Status. In: Proceedings of the AMIA Symposium, p. 662. American Medical Informatics Association (2001)
20. Suárez-Figueroa, M.C., Gómez-Pérez, A., Motta, E., Gangemi, A.: Ontology engineering in a networked world. Springer (2012)

21. Suárez-Figueroa, M.C., Pradel, C., Hernandez, N.: Verifying ontology requirements with SWIP. In: EKAW 2012 (2012)
22. Thomas, E., Pan, J.Z., Ren, Y.: TrOWL: Tractable OWL 2 Reasoning Infrastructure. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) ESWC 2010, Part II. LNCS, vol. 6089, pp. 431–435. Springer, Heidelberg (2010)
23. Uschold, M., Gruninger, M.: et al. Ontologies: Principles, methods and applications. Knowledge Engineering Review 11(2), 93–136 (1996)
24. Zemmouchi-Ghomari, L., Ghomari, A.R.: Translating natural language competency questions into SPARQL queries: A case study. In: WEB 2013, pp. 81–86 (2013)