Generating Answerable Questions from Ontologies for Educational Exercises: *Approaches*

Toky Raboanary, Steve Wang, and C. Maria Keet

Department of Computer Science, University of Cape Town, South Africa
traboanary@cs.uct.ac.za, WNGSHU003@myuct.ac.za, mkeet@cs.uct.ac.za

1 Introduction

We aim to generalise the generation of educational questions from ontologies. This document focuses on the presentation of the three approaches that we have designed, implemented and evaluated for the dynamic question generation:

- template variables using foundational ontology categories (Appr 1),
- using main classes from the domain ontology (Appr 2), and
- sentences mostly driven by natural language generation techniques (Appr 3).

The overview of the three approaches is presented in Figure 1. Appr 1 and Appr 2 adopt 'Algorithm 1', with the difference that the former takes Type A and Type B templates as input and the latter takes Type C templates as input. Appr 3 uses 'Algorithm 2' and takes Type D templates as input. The differences between the Type A, Type B, Type C and Type D templates can be found in the file: *QuestionTypesTemplates.pdf*. Algorithm 1 uses only OWL API [3] and Hermit Reasoner [2], whereas Algorithm 2 also uses SimpleNLG [1] and WordNet [4].

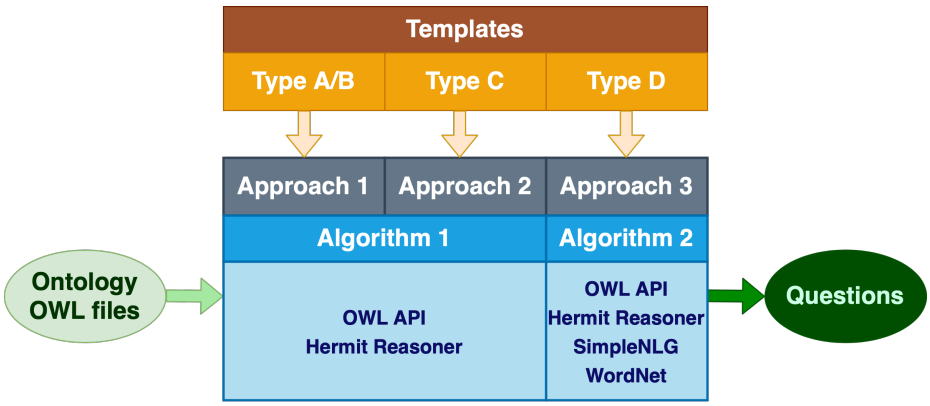

Fig. 1: Overview of the three approaches.

2 Algorithm 1: ontology element-based templates

Algorithm 1 is composed of some variant sub-algorithms depending on the type of questions, but several steps are the same. There are 3 different types of tokens that are going to replace the slots in templates: quantifier tokens (denoted with [quantifier]), OWLObjectProperty tokens, and OWLClass tokens. A [quantifier] in the template is replaced with either 'some' ($\exists$) or 'only' ($\forall$). When the token appears as an [ObjectProperty] then it can be replaced with any of its object sub-properties in the ontology that satisfies the axiom prerequisites of the question type. If [X], indicating an OWLClass, appears in the template, then it can be replaced with any subclass of X.

Overall, the algorithm picks a template and tries to fill it with contents from the ontology, taking into account the vocabulary, axiom prerequisites, hyphen checking (e.g., 'Bumble-Bee' is converted to 'bumble bee') and article checking (e.g., 'a elephant' is converted to 'an elephant'). For example, with the template "*Does a* [Thing] [ObjectProperty] *a* [Thing]*?*", the algorithm can generate a question like "Does a catalogue describe a collection?" from the axiom $\mathsf{Catalogue} \sqsubseteq \exists \mathsf{describes.Collection}$.

Algorithm 1

```
1: Select a question template for a question type
2: Get tokens from template
3: for each token in tokens do
4:    if isObjectProperty(token) == true then
5:       Select Randomly an Element of SubProperty following the axiom prerequi-
         sites for the selected question type using Reasoner
6:       checkFormat(token) {hyphen}
7:    else if isClass(token) == true then
8:       Select Randomly an Element of SubClass following the axiom prerequisites
         for the selected question type using Reasoner
9:       checkFormat(token) {hyphen}
10:      checkArticle(token)
11:   else if isQuantifier(token) == true then
12:      Select a Random Quantifier either 'only' or 'some' following the axiom pre-
         requisites for the selected question type using Reasoner
13:   end if
14: end for
15: Fill in the template
16: Render
```

It is implemented using Java and uses the OWL API [3] to access the ontology and the Hermit Reasoner [2] for recursively fetching vocabulary.

3 Algorithm 2: natural language-driven templates

Algorithm 2 does not just fill in the question templates, but from a selected type of questions, fetches all axioms following the related axiom prerequisites, processes the contents of the ontology by fetching the vocabulary elements of a selected axiom, picks an appropriate variant of a template that the vocabulary can be used in, and makes some linguistic adaptation before generating the whole question. The improvements incorporated were partially informed by the analysis of the sentences generated by Algorithm 1 that were evaluated as 'bad' questions.

Algorithm 2 receives as input the type of questions that we want to generate, the concerned ontology and the set of templates. Precisely, the templates used by this algorithm is Type D templates (explained in *QuestionTypesTemplates.pdf*). It fetches all asserted and inferred axioms in the ontology that satisfies the axiom prerequisites for the considered type of questions by using Hermit Reasoner [2]. For instance, if the type of questions: "True/False Questions with additional quantifier" (e.g. of template: "*True or false: A* X OP *some* Y") is selected, all axioms following the axiom pattern $\mathsf{X} \sqsubseteq \exists \mathsf{OP}.\mathsf{Y}$ (Answer: True) and $\mathsf{X} \sqsubseteq \neg\exists \mathsf{OP}.\mathsf{Y}$ (Answer: False) are fetched. Then, one of them is randomly selected for the generation of the sentence. However, instead of selecting it randomly, one could choose an important axiom(s) by choosing relevant classes or object property if needed.

The next step is to fetch the vocabulary elements of the selected axiom. Then, our algorithm check the category of the Object Property of the selected axiom in order to choose the appropriate template for a given axiom by considering the POS of classes and OPs, and classifying the OP. We designed an algorithm based on an Finite State Machine (FSM) that classifies the name given to an OP to find the appropriate template for an axiom, and provides the appropriate equivalent text. The algorithm considers 6 categories, listed and illustrated in Table 1. An OP name may: 1) have a verb, 2) start with a verb followed by a preposition, 3) start with 'has' and followed by nouns, 4) be composed of 'is', nouns and a preposition, 5) start with 'is', followed by a verb in a past participle form and ends with a preposition, or 6) start with 'is', followed by a verb in a past participle form and ends with 'by' (i.e., passive voice variants for 4-6). The FSM strategy is a sequence detector to determine the category of an OP and chunks it. For instance, the OP *is-eatenBy*, which is an instance of OP_Is_Past_Part_By (the 6^{th} variant), is transformed into a list of words (is, eaten, by). Then, it detects each component, and from that, the POS of each token is obtained, and, finally, it generates the appropriate group of words: "is eaten by", which will be used in the question. So, for the axiom $\mathsf{Leaf} \sqsubseteq \exists \mathsf{eaten\text{-}by}.\mathsf{Giraffe}$, the appropriate template is "*Is a* [T_Noun][OP_Is_Past_Part_By] *a* [T_Noun]*?*" and a correct generated question would be "Is a leaf eaten by a giraffe?" rather than "Does a leaf eaten by a giraffe?".

Also, the algorithm checks the format of each vocabulary element such as camel case, hyphen and underscore. Then, suppose the vocabulary element is a class. In that case, it considers its POS such as noun and infinitive verb, checks

Algorithm 2

Input: type of questions, ontology, set of templates
Fetch all asserted and inferred **axioms** that satisfies the **axiom prerequisites** for the selected **type of questions** using Reasoner
Select **one axiom** randomly
Fetch the **vocabulary elements** *vocabularyelements* of the selected axiom
Check the **category of the Object Property (OP)** of the selected axiom (Table 1 presents the categories and some examples)
Get the **appropriate template** and **formatted vocabulary element** ($vocOP$) with respect to the type of questions, the selected axiom and the category of the OP from the OP classification {The OP Classifier algorithm (included in the supplementary material) uses **SimpleNLG** to obtain the third-person singular in the present simple tense and the past participle form of a verb, and **WordNet** to check the POS (Part-Of-Speech) of words such as verb, noun, adjective and preposition)}
for each *voc* **in** *vocabularyelements* **do**
    voc = **checkFormat**(voc) {camel case, hyphen, underscore}
    if isClass(voc) == $true$ **then**
        if POS(voc) == $noun$ **then**
            if isCountableNoun(voc) == $true$ **then**
                $formattedVoc$ = **addArticle**(voc, "a"/"an"/"") {Choose the right article a/an/(empty) for voc by using **SimpleNLG**, if the axiom contains an existential quantifier, do not insert any article}
            else
                $formattedVoc$ = voc {$formattedVoc$ is the formatted vocabulary element to fill in the appropriate template}
            end if
        else if POS(voc) == $infinitive\_verb$ **then**
            $formattedVoc$ = **gerundForm**(voc) {By using **SimpleNLG**}
        else
            $formattedVoc$ = voc {No transformation if voc is neither a noun nor an infinitive verb}
        end if
    else if isQuantifier(voc) == $true$ **then**
        $formattedVoc$ = **addQuantifier**(voc, "some"/"only") {for $\exists$, use "some"; and for $\forall$, use "only"}
    else if isObjectProperty(voc) == $true$ **then**
        $formattedVoc$ = voc {No further transformation, here, voc = $vocOP$; it is already formatted by the Object Property Classifier)}
    end if
    Fill in the selected **appropriate template** with $formattedVoc$ {the formatted vocabulary element}
end for
Output: Question

Table 1: Object Property classification with illustrative examples

Object property	**Category**	**Equivalent text**
eats	OP_Verb	eats
participate-in	OP_Verb_Prep	participates in
hasState	OP_Has_Nouns	has a state
isContiguousPortionOf	OP_Is_Nouns_Prep	is a contiguous portion of
containedIn	OP_Is_Past_Part_Prep	is contained in
eaten-by	OP_Is_Past_Part_By	is eaten by

if it is a countable noun or not and renders the appropriate formatting. If the vocabulary element is a quantifier, it uses 'some' or 'only' if the quantifier is '$\exists$' or '$\forall$', respectively. Then, if the vocabulary element is an object property, the rendering is based on the Object Property Classifier (its pseudo-code is presented in the file: Algo OP Classifier.pdf).

Finally, Algorithm 2 generates the appropriate question.

Regarding implementation, OWL API [3], Hermit Reasoner [2], SimpleNLG [1], and WordNet [4] are used by the algorithm to generate questions.

References

1. Gatt, A., Reiter, E.: SimpleNLG: A realisation engine for practical applications. In: Proc. of ENLG'09. pp. 90–93 (2009)
2. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z.: HermiT: an OWL 2 reasoner. J. Autom. Reas. **53**(3), 245–269 (2014)
3. Horridge, M., Bechhofer, S.: The OWL API: A java API for OWL ontologies. Semantic Web **2**(1), 11–21 (2011)
4. Miller, G.A.: Wordnet: a lexical database for english. Comm. of the ACM **38**(11), 39–41 (1995)