Natural Language Generation From Ontologies

Van Nguyen, Son Tran, and Enrico Pontelli

New Mexico State University, Las Cruces, NM 88003, USA
{vnguyen,tson,epontell}@cs.nmsu.edu

Abstract. This paper addresses the problem of automatic generation of *natural language descriptions* for *ontology-described artifacts.* The original motivation for the work is the challenge of providing textual narratives of automatically generated scientific workflows (e.g., paragraphs that scientists can include in their publications). The paper presents two systems which generate descriptions of sets of atoms derived from a collection of ontologies. The first system, called `nlgPhylogeny`, demonstrates the feasibility of the task in the *Phylotastic* project, providing evolutionary biologists with narrative for automatically generated analysis workflows. `nlgPhylogeny` utilizes the fact that the *Grammatical Framework (GF)* is suitable for the natural language generation (NLG) task; the paper shows how elements of the ontologies in Phylotastic, such as web services and information artifacts, can be encoded in GF for the NLG task. The second system, called $\texttt{nlgOntology}^A$, is a generalization of `nlgPhylogeny`. It eliminates the requirement that a GF needs to be defined and proposes the use of *annotated ontologies* for NLG. Given a set of annotated ontologies, $\texttt{nlgOntology}^A$ generates a GF suitable for the creation of natural language descriptions of sets of atoms derived from these ontologies. The paper describes the algorithms used in the development of `nlgPhylogeny` and $\texttt{nlgOntology}^A$ and discusses potential applications of these systems.

Keywords: Natural Language Generation · Ontologies · Web service · Grammatical Framework · Attempto Controlled English.

1 Introduction

In many application domains, where users are not proficient in computer programming, it is of the utmost importance to be able to communicate the results of a computation in an easily understandable way, e.g., using text rather than a complex data structure or mathematic formulae. The problem of generating *natural language explanations* has been explored in several research efforts. For example, the problem has been studied in the context of question-answering systems,[1] recommendation systems,[2] etc. With the proliferation of spoken dialogue systems and conversational agents, verbal interfaces such as Amazon Echo and Google Home for human-robot-interaction, and the availability of text-to-speech

[1] http://coherentknowledge.com
[2] http://gem.med.yale.edu/ergo/default.htm

programs, such as the TTSReader Extension,[3], the application arena of systems capable of generating natural language representations will continue growing. In this paper, we describe two systems for generating natural language descriptions of collections of atoms derived from a set of ontologies.

The first system, called `nlgPhylogeny`, is used to generate natural language descriptions of automatically generated workflows, obtained by composing web services. This is motivated by the needs of the *Phylotastic* project [1]; the project provides tools for the automated construction of workflows that allow evolutionary biologists, teachers, and students to extracts phylogenies relevant to given sets of species. The automated construction of workflows is justified by the existence of a large number of web services that perform parts of a desired analysis protocol, and the complexity of effectively interfacing the services (e.g., due to the lack of data format standardization). A typical Phylotastic workflow is composed of operations to collect list of species names (e.g., from a scientific paper), "clean" them to ensure that the proper scientific names are used, extract a subtree of a reference phylogeny that covers the desired species and visualize it. Phylotastic has been implemented using an *Answer Set Programming (ASP)* backend for reasoning about ontologies and for web service composition [6]. The web services are described by an ontology, the Phylotastic ontology (*PO*). *PO* is composed of two parts: an ontology that describes the artifacts manipulated by the services (e.g., alignments, phylogenetic trees, species names) [7] and an ontology to describe the operations performed by the services (the *WSO*).

Figure 2 displays a sample output of `nlgPhylogeny` given the workflow in Figure 1. The workflow in this example is a plan generated by the ASP-based web service composition component of the Phylotastic project [6], and consists of a sequence of steps (green rectangles). The boxes before and after each green rectangle represent input(s) and output(s) of the service, respectively. As the inputs of one service might require some format different from the format of the previous outputs, data conversions might be necessary (the double arrows). Each step corresponds to a processing step on data provided by one of the preceding steps. Specifically, the workflow is composed of three steps:

- Extracting the set of organisms from the input text;
- Resolving the names of the identified organisms (e.g., correct spelling, identify proper scientific names); and
- Deriving the corresponding phylogenetic tree.

Figure 2 shows the description of the workflow as generated by `nlgPhylogeny`. Since the fact that to illustrate a workflow, an graphical version is approximately good enough, but to put the workflow in a biological paper, sometimes, authors would need to write some explanations for the workflow. We find that it would be helpful to generate the textural version as a complement to the graphical version, and provide them as a package. So, the authors are free to choose which versions to include in their paper. Moreover, we recognize that our general idea can be a bridge between ontology developers and ontology users or engineers who use the ontology in question-answering system. While ontology developers

[3] https://ttsreader.com

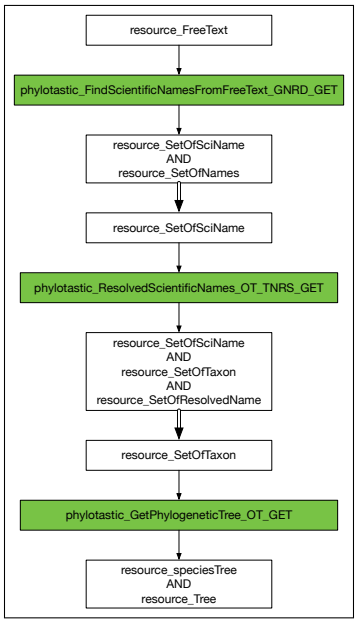

Fig. 1. An automatically generated workflow

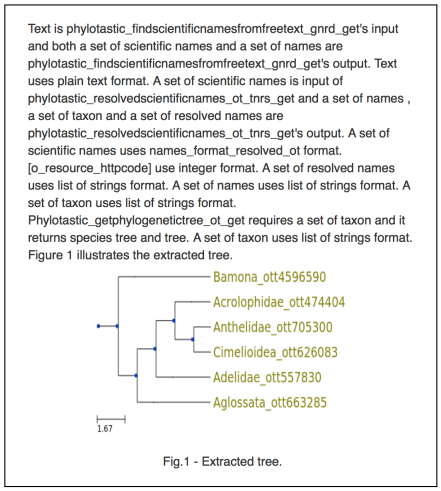

Fig. 2. Description generated for workflow in Fig. 1 by `nlgPhylogeny`

just need to add a little more annotations on their work, the benefit for ontology users is huge because they will no longer need to develop the answering module from scratch. The answer generated from our idea will mimic the grammar structure of annotations provided by ontology developers, but different in content corresponding to the queried data.

As discussed in detail later, `nlgPhylogeny` exploits the NLG capabilities of the *Grammatical Framework (GF)* [8]. This requires the development of a GF for the entities in the Phylostatic projects (described by the ontology). For small ontologies, the manual development of the GF for the NLG task is feasible, but it is an improbable task for large ontologies. Furthermore, `nlgPhylogeny` will not be applicable for other ontologies. It is, however, feasible to consider a situation where an ontology engineer has the necessary domain knowledge to explicitly add meta-information to the concepts as they are progressively added to the ontology. The `nlgOntology`^{A} system demonstrates that, as long as meta-information is added in the ontology following proper guidelines, it is possible to generate the description for the atoms derived from annotated ontologies without the manual creation of a GF.

The project critically relies on logic programming. ASP is employed by the composition system and to manage the connection with the ontology. The Attempto Parsing Engine is available in GitHub [4] and it is written in SWI-Prolog.

[4] https://github.com/Attempto/APE

The program to convert lexicon extracted from annotations in the ontologies to lexicon to generate the GF concrete syntax is also a Prolog-based program.

The rest of the paper is organized as follows. We begin with a brief review of the *Grammatical Framework* and *Attempto Controlled English,* the two frameworks used in this paper. The following two sections describe `nlgPhylogeny` and $\texttt{nlgOntology}^A$, respectively. We conclude the paper with a discussion of potential uses of $\texttt{nlgOntology}^A$ and of the proposed technologies developed in this paper.

2 Background

2.1 Grammatical Framework

The Grammatical Framework (GF) [8] is a system used for working with grammars; it is composed of a programming language used to design grammars along with a theory about grammars and languages. GF also comes with a GF Resource Grammar Library and a GF runtime API for working with GF programs.

A GF program has two main parts. The first part is the *Abstract syntax* which defines what meanings can be expressed by a grammar. The abstract syntax defines categories (i.e., types of meaning) and functions (i.e., meaning-building components). The following is an example of an abstract syntax component:

```
abstract Food = {
        flags   startcat = Phrase ;
        cat
                Phrase ; Item ; Kind ; Quality ;
        fun
                Is : Item -> Quality -> Phrase ;
                This : Kind -> Item ;
                QKind : Quality -> Kind -> Kind ;
                Cheese, Fish : Kind ;
                Very : Quality -> Quality ;
                Warm, Italian, Delicious : Quality ;
}
```

In this syntax, `Phrase`, `Item`, `Kind` and `Quality` are types of meanings. The `startcat flag` declaration states that `Phrase` is the default start category for parsing and generation. `Is` is a function accepting two parameters, of type `Item` and `Quality`. This function returns a meaning of `Phrase` category.

The second part is composed of *one or more concrete syntax specifications.* Each concrete syntax defines the representation of meanings in each output language. For example, the corresponding concrete syntax that maps functions in the `abstract Food` grammar above to strings in English is:

```
concrete FoodEng of Food = {
  lincat
    Phrase, Item, Kind, Quality = {s : Str} ;
  lin
    Is item quality = {s = item.s ++ "is" ++ quality.s} ;
    This kind = {s = "this" ++ kind.s} ;
```

```
    QKind quality kind = {s = quality.s ++ kind.s} ;
    Cheese = {s = "cheese"} ;
    Fish = {s = "fish"} ;
    Very quality = {s = "very" ++ quality.s} ;
    Warm = {s = "warm"} ;
    Italian = {s = "Italian"} ;
    Delicious = {s = "delicious"} ;
}
```

In this concrete syntax, the linearization type definition (`lincat`) states that `Phrase`, `Item`, `Kind` and `Quality` are strings (`s`). Linearization definitions (`lin`) indicate what strings are assigned to each of the meanings defined in the abstract syntax. Various types of linearization type definitions are considered in GF (e.g., string, table). Some functions represent a simple string but some functions (e.g., `Is` or `This`) defines a concatenation of strings.

Intuitively, each function in the abstract syntax represents a rule in a grammar. The combination of rules used to construct a meaning type can be seen as a syntax tree. The visualization of the tree representing the `Phrase` *"this delicious cheese is very Italian"* is illustrated in Figure 3.

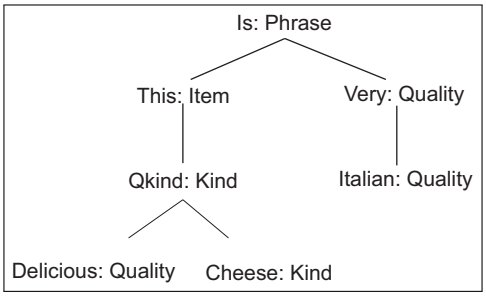

Fig. 3: Example syntax tree

GF has been used in a variety of applications, such as query-answering systems, voice communication, language learning, text analysis and translation, and natural language generation [9, 3]. GF has been used extensively in automated translation and it is the main vehicle behind the MOLTO project, that aims at developing a set of tools for high-quality and real-time translation of text between multiple languages[5]. To see how it works, let us augment our program with a concrete syntax for Italian as follows:

```
concrete FoodIta of Food = {
  lincat
    Phrase, Item, Kind, Quality = {s : Str} ;
  lin
    Is item quality = {s = item.s ++ "e'" ++ quality.s} ;
    This kind = {s = "questo" ++ kind.s} ;
    QKind quality kind = {s = kind.s ++ quality.s} ;
    Cheese = {s = "formaggio"} ;
    Fish = {s = "pesce"} ;
    Very quality = {s = "molto" ++ quality.s} ;
    Warm = {s = "caldo"} ;
    Italian = {s = "italiano"} ;
    Delicious = {s = "delizioso"} ;
}
```

The translation from English to Italian can be performed as follows in the GF API:

[5] http://www.molto-project.eu

```
> parse -lang=FoodEng "this fish is warm" | linearize -lang=FoodIta
    questo pesce e' caldo
```

We use a pipe which includes the `parse` and `linearize` commands to find the syntax tree of the sentence *"this fish is warm"* then turn that tree into a `FoodIta` sentence. The last line is the result of the translation process. The translation process is very similar to currency exchange in the old days, when exchange was done only in gold. Assume we want to exchange US Dollars for Euros; we first exchange US Dollars for gold, then, exchange gold for Euros. Correspondingly, in GF the intermediate result in the translation process is the syntax tree which contains the meaning of the translated sentence.

2.2 Attempto Controlled English

A GF program produces sentences whose syntax is specified by its abstract syntax; this structure also determines the quality of its output. Developing a GF syntax (abstract or concrete) requires understanding functional programming; this is a level of knowledge that might not be suitable for users who are not familiar with programming—as is the case of biologists using Phylotastic to create and execute phylogenetic workflows. As we will see in the next section, our `nlgPhylogeny` system can utilize GF to generate descriptions of Phylotastic workflows. It requires, however, a considerable amount of domain-specific knowledge. To alleviate this problem, we investigate a combination of annotated ontologies and the *Attempto Controlled English (ACE)* [4] for the same task, which results in the system `nlgOntology`^{A}.

ACE is a controlled natural language, i.e., a subset of standard English with a restricted syntax and restricted semantics, described by a small set of construction and interpretation rules. ACE sentences are normal English sentences and can be read and understood by any English speaker. However, ACE is a formal language that can be used for knowledge representation; ACE texts are computer-processable and can be unambiguously translated into discourse representation structures, a syntactic variant of first-order logic. An ACE grammar consists of construction rules for both simple and composite sentences, interrogative and imperative sentences. ACE can be encoded in GF and used for NLG.

3 Generating Sentences from GF

In this section, we describe the `nlgPhylogeny` system. Figure 4 shows the overall architecture of `nlgPhylogeny`. The main component of the system is the *GF generator* whose inputs are the Phylotastic ontology and the elements necessary for the NLG task (i.e., the set of linearizations, the set of pre-defined conjunctives, the set of vocabularies, and the set of sentence models). The output of the *GF generator* is a GF program, i.e., a pair of GF abstract and concrete syntax. This GF program is used for generating the descriptions of workflows via the GF runtime API. The adapter provides the GF generator with the information from the ontology, such as the classes, instances, and relations. We will describe in more details the elements of `nlgPhylogeny`in Sect. 3.2.

3.1 Web Service Ontology (WSO)

Phylotastic uses web service composition to generate workflows for the extraction/construction of phylogenetic trees. It makes use of two ontologies: WSO and PO. WSO

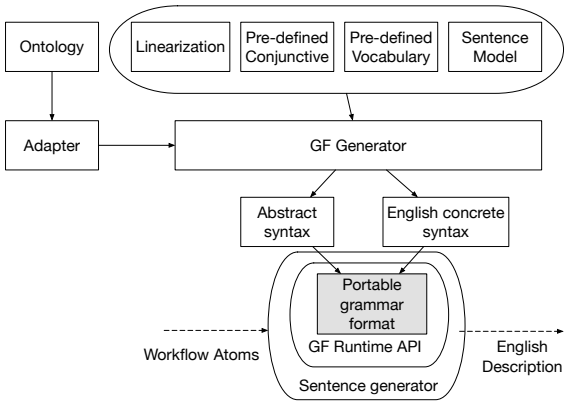

Fig. 4. Overview of `nlgPhylogeny`.

encodes information about the registered web services, classified in a taxonomy of classes of services. In the following discussion, we refer to a simplified version of the ASP encoding of the ontologies used in [6], to facilitate readability.

In WSO, a service has a name and is associated with a list of inputs and outputs. For example, the service named *FindScientificNamesFromWeb_GET* in the ontology is an instance of the class *names_extraction_web*. The outputs and inputs of *FindScientificNamesFromWeb_GET* are encoded by the three atoms:

```
has_input(FindScientificNamesFromWeb_GET,resource_WebURL,url_format).
has_output(FindScientificNamesFromWeb_GET,resource_SetOfSciName,
          scientific_names_format).
has_output(FindScientificNamesFromWeb_GET,resource_SetOfNames,
          list_of_strings).
```

In the above atoms, the first argument is the name of the service, the second is the service input or output, and the last argument is the data type of the second argument.

The web service ontology of the Phylotastic project is exported to an ASP program (from its original OWL encoding) and enriched with a collection of ASP rules to draw inferences about classes, inheritance, etc. `nlgPhylogeny` employs these rules to identify information related to the set of atoms whose description is requested by a user—e.g., What are the inputs of a service? What is the data type of an input x of a service y?

3.2 GF generator

Each Phylotastic workflow is an acyclic directed graph, where the nodes are web services, each consumes some resources (inputs) and produces some resources (outputs). An example of the specification of a workflow is as follows.[6]

[6] For simplicity, we use examples which are linear sequences of services. We also trim the names of services for readability.

```
occur_concrete(GenerateGeneTree_From_Genes,0).
occur_concrete(ExtractSpeciesNames_From_Gene_Tree_GET,1).
occur_concrete(GeneTree_Scaling,2).
occur_concrete(ResolvedScientificNames_OT_TNRS_GET,3).
```

This set of atoms is a partial description of the result of a web service composition process, as described in [6]. Intuitively, this set of atoms represents a plan consisting of 4 steps. At each step, a concrete instance of the service class named by the first argument of the atom `occur_concrete/2` is executed.

To generate the description of a workflow, we adapt the general theoretical framework proposed in [10]. This framework consists of three major processing phases: **(1)** Document planning (content determination), **(2)** Microplanning, and **(3)** Surface realization. The document planning phase is used to determine the structure of the text to be generated. Based on the structure determined in the document planning phase, the microplanner makes lexical/syntatic choices to generate the content of the sentences, and the realization phase generates the actual sentences. In our work, we combine the microplanning and surface realization phase into a single phase due to the nature of the grammar definition and the capability of GF in sentence generation.

In the document planning step, we create, for each occurrence atom, a sentence which specifies the input(s) and output(s) of the service mentioned in the first argument of the atom. Optionally, users can choose to describe the service in more details, one or two more sentences about the data type of the service's inputs or outputs can be included. As we have mentioned in the previous subsection, the information about the inputs, outputs, and data types of the inputs and outputs of a service can be obtained via the ASP reasoning engine of the Phylotastic system. In general, we identify the document planning structure described in Table 1.

message 1	
argument_1:	instance or class in ontology
argument_2:	list of service inputs
argument_3:	list of service outputs
message 2 **(optional)**	
argument_1:	name of input or output of service
argument_2:	data type of argument_1
message 3 **(optional)**	
argument:	actual data involved in the workflow

Table 1. Document Planning Structure

The document planning phase determines three messages for the sentence generation phase. Each message will be constructed using the arguments as mentioned in Table 1. While the first message is mandatory, the other two messages are optional.

In the microplanning step, we focus on developing a GF generator that can produce a portable grammar format (**pgf**) file [2]. This file is able to encode and generate 3 types of sentences as mentioned above. The GF generator (see Fig. 4) accepts two flows of input data. The first one is the flow of data from the ontology, which is maintained by an adapter. The *adapter* is the glue code that connects the ontology to the GF generator. Its main function is to extract classes and properties from the ontology.

The second flow is the flow of data from predefined resources that cannot be automatically obtained from the ontology—instead they require manual effort from both the ontology experts and the linguistic developers.

- A list of *linearizations:* the translations of ontology entities into linguistic terms. This translation is performed by experts who have knowledge of the ontology domain. An important reason for the existence of this component is that some classes or terms used in the ontology might not be directly understandable by the end user. This may be the result of very specialized strings used in the encoding of the ontology. For example, the class *phylotastic_ResolvedScientificNames_OT_TNRS_POST* can be meaningfully linearized to *Name Resolution service provided by OpenTree* in Phylotastic ontology.
- Some *sentence models* which are principally Grammatical Framework syntax trees with meta-information. The meta-information denotes which part of syntax tree can be replaced by some *vocabulary* or *linearization*. As indicated above, we decided that each occurrence atom in a workflow will be described by at most three sentences. For example, if we consider the first message in the document planning structure, the generated sentence will have the inputs and the outputs of a service; the second message indicates a sentence about the data type of its first argument (input or output); the third message is about the actual data used during the execution of the workflow. However, the messages do not specify how many inputs and outputs should be included in the generated sentence. This means that sentences have different structures, i.e., the structure of a sentence representing a service that requires one input and one output is different from the structure of a sentence representing a service that does not require any inputs. These variations in sentences are recorded in the *model sentence* component.
- A list of *pre-defined vocabularies* which are domain-specific components for the ontology. A *pre-defined vocabulary* is different from linearizations, in the sense that some lexicon may not be present in the ontology but might be needed in the sentence construction. The predefined vocabulary is also useful to bring variety in word choices when parts of a *model sentence* are replaced by the GF generator. For example, we would not want the system to keep generating a sentence of the form *"The service A has input X"* given an atom of the form `occur_concrete`(A, T), but sometimes *"The service A requires input X",* or *"The service A needs input X",* etc. To achieve this, we keep "have", "require" and "need" in the set of *pre-defined vocabularies* and randomly select a verb to replace the verb in *model sentence*.
- A configuration of *pre-defined conjunctives*, which depend on the document planning result. Basically, this configuration defines which sentences accept a conjunctive adverb in order to provide generated text transition and smoothness.

To encode sentences, the GF generator defines 3 categories: Input, Output and Format in the abstract syntax. The corresponding English concrete syntax is as follows:

```
concrete PhyloEng of Phylo = open SyntaxEng, ParadigmsEng, ConstructorsEng
      in {
            lincat
                Message = S; Input = NP; Output = NP; Format = NP;
          ... }
```

`SyntaxEng`, `ParadigmsEng`, `ConstructorsEng` are GF Resources Grammar libraries[7] providing constructors for sentence components like Verb, Noun Phrase, etc. in English.

The GF generator obtains information about the services (e.g., how many inputs/outputs has the service? what are the data types of the inputs/ouputs? etc.) by query-

[7] http://www.grammaticalframework.org/lib/doc/synopsis.html

ing the ontology (via the adapter). Based on the number of inputs and outputs of a service, the GF generator determines how many parameters will be included in the GF abstraction function corresponding to the service. Furthermore, for each input or output of a service, the GF generator includes an *Input* or *Output* in the GF abstract function. For example, the encoding of $occur\_concrete(FindScientificNamesFromWeb\_GET, 1)$ in the GF abstract syntax is

```
f_FindScientificNamesFromWeb_GET: Input -> Output -> Message;
i_resource_WebURL: Input;
o_resource_SetOfNames: Output;
```

Next, the GF generator looks up in the *sentence models* a model syntax tree whose structure is suitable for the number of inputs and outputs of the service. If such syntax tree exists, the GF generator will replace parts of the syntax tree with the GF service input and output functions, to create a new GF syntax tree which can be appended to the GF concrete function. The functions in the abstract syntax correspond to the following functions in the GF concrete syntax:

```
f_phylotastic_FindScientificNamesFromWeb_GET i_resource_WebURL
o_resource_SetOfNames = mkS and_Conj
    (mkS (mkCl phylotastic_FindScienticNamesFromWeb_GET_in
               (mkV2 "require") i_resource_WebURL))
    (mkS (mkCl phylotastic_FindScienticNamesFromWeb_GET_out
               (mkV2 "return" ) o_resource_SetOfSciName ));
i_resource_WebURL = mkNP(mkCN (mkN "webURL"));
i_resource_SetOfNames = mkNP(mkCN (mkN "asetof names"));
```

The above functions consist of several syntactic construction functions which are implemented in the GF Resources Grammar:

- `mkN` which creates a noun from a string;
- `mkCN` which creates a common noun from a noun;
- `mkNP` which creates a noun phrase from a common noun;
- `mkV2` which creates a verb from a string;
- `mkCl` which creates a clause. A clause can be constructed from sequence of a noun phrase, a verb and another noun phrase (NP V2 NP);
- `mkS` which creates a sentence. A sentence can be constructed from a clause (Cl) or from 2 other sentences and a conjunction word (and_Conj S S).

From the abstract and concrete syntax specifications built by the GF generator, the atom

`occur_concrete(phylotastic_FindScientificNamesFromWeb_GET,1)`

is translated into the sentence

> *The input of phylotastic_FindScientificNamesFromWeb_GET is a web link, and its outputs are a set of species names and a set of scientific names.*

We use the same technique to encode the other types of sentences indicated by the document planning structure. This is how the GF generator has been implemented. Figure 1 is an example output of the current version of `nlgPhylogeny`.

4 Automatic Natural Language Generation From Annotated Ontology: `nlgOntology`^{A}

The previous section shows that, with sufficient knowledge about the ontology and pre-defined descriptions about elements in the ontology, we can utilize the current technology in NLG to generate a description of a set of atoms derived from the ontology. It also highlights that the process requires manual labor and domain expertise. Such approach is feasible only in small ontologies related to uncomplicated grammars and elementary lexicons. The application of the same process to medium or large ontologies is likely to be too costly or time consuming. On the other hand, we can observe that ontologies often include meta-data encoding of their elements. Furthermore, information extracted from the meta-data of an ontology is often sufficient for a basic understanding of the concepts that can be derived from the ontology. Motivated by this observation, we develop an automatic natural language generation method for ontologies whose meta-data can be understood by an ACE parser. We will refer to ontologies satisfying this assumption simply as *annotated ontology*. A simple annotated ontology is as following.

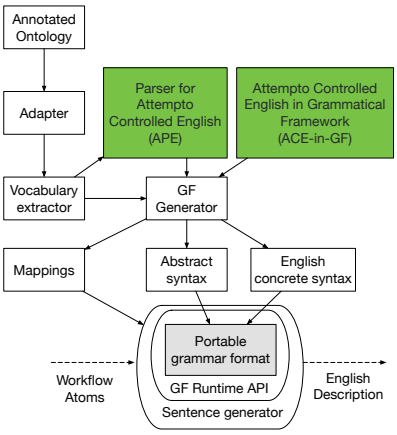

Fig. 5: Overview of `nlgOntology`^{A}

```
%% @n: Company
class(Com)

%% @pn: Apple_Inc
instanceOf(Com, Apple)

%% @pn: Beats
instanceOf(Com, Beats)

%% @lin: Beats is a company of Apple_Inc
own(Apple, Beats)

%% @pn: Silicon_Grail_Corp_Chalice
instanceOf(Com, Sgcc)

%% @pn: Silicon_Grail
instanceOf(Com, Sg)

%% @lin: Apple_Inc acquires Beats
acquire(Apple, Beats)
```

In the above ontology, `Com` is a class, `Apple`, `Beats`, `Sgcc` and `Sg` are instances of the class `Com`, and `acquire` and `own` are two properties. The tags

- `%% @n` marks a noun
- `%% @pn` denotes a proper noun
- `%% @lin` signals a translation of an atom to an Attempto English sentence

Ontologies annotated in this way can be understood by `nlgOntology`^{A}. We now describe the `nlgOntology`^{A} system.

4.1 Overall Architecture

Figure 5 shows the overall structure of `nlgOntology`^{A}. The GF generator described in Figure 5 uses data and functions from three main components:

- A *vocabulary extractor*, which is responsible for collecting nouns, proper nouns, adjectives and verbs from the ontology. The *vocabulary extractor* also creates a mapping of classes or instances in the ontology to their linearizations. Moreover, in the case of adjectives and verbs, the *vocabulary extractor* will query some vocabulary dictionaries to collect information like type of verbs (transitive, intransitive) and verbs in different forms (finite singular, infinite, etc.).
- The *Attempto Controlled English Parser (APE),* which analyzes sentences extracted from the ontology. The parser translates ACE text into *discourse representation structures (DRS)* [5].
- *Attempto Controlled English in Grammatical Framework (ACE-in-GF),* which is an implementation of the Attempto Controlled English grammar in the Grammatical Framework syntax.

The outputs of the generator are a portable grammar format (**pgf**) file, a mapping of annotated atoms in the ontology into GF syntax trees, and a mapping of concepts used in the ontology into GF functions. These data will be used in the re-construction sentence progress which is described next.

Algorithm 1 Generation of portable grammar format

Require: annotated ontology, some annotations are ACE parable sentences
1: $n \leftarrow$extract nouns and proper nouns from ontology
2: $s \leftarrow$extract sentences from ontology
3: $a \leftarrow$ *empty*, $v \leftarrow$ *empty*
4: add n to APE lexicon
5: **for** i in n **do**
6: $n', a', v' \leftarrow$parse i using APE
7: $n := n \cup n'$, $a := a \cup a'$, $v := v \cup v'$
8: **end for**
9: **for** i in n **do**
10: find singular and plural form of i
11: **end for**
12: **for** i in a **do**
13: find comparative and supercomparative form of i
14: **end for**
15: **for** i in v **do**
16: find transitive and intransitive form of i
17: **end for**
18: generate vocabulary, generate mappings
19: convert vocabulary to GF syntax
20: compile grammar in ACE-in-GF and generated syntax

4.2 Generation of Portable Grammar Format

To generate the **pgf** file, the GF generator performs the procedure shown in Procedure 1. Lines 1-2 extract annotations from the ontology. Lines 3 initialize variables holding adjectives and verbs. Line 4 enriches the APE lexicon with the nouns and proper nouns. This allows the APE to recognize proper nouns that are possibly present in the sentences extract in s. Furthermore, it helps increase the accuracy when a sentence is parsed by APE. Next, the for-loop in lines 5-8 iterates through all sentences to collect

new lexicon. Lines 9-17 find all possible forms of words. Line 18 creates the vocabulary file and mapping file from information obtained from previous steps. The vocabulary file is written in Prolog. For example, a portion of the vocabulary file extracted from the annotations from the ontology in the beginning of this section looks as follows:

```
noun_pl('companies', company, neutr).
noun_sg('company', company, neutr).
pn_sg('Apple_Inc', 'Apple_Inc', neutr).
pn_sg('Beats', 'Beats', neutr).
pn_sg('Silicon_Grail_Corp_Chalice', 'Silicon_Grail_Corp_Chalice', neutr).
pn_sg('Silicon_Grail', 'Silicon_Grail', neutr).
tv_finsg(acquires, acquire).
iv_finsg(is, be).
```

Line 19 converts the vocabulary file to GF syntax. As an example, the conversion produces the GF concrete syntax file:

```
lin
    company_N = aceN "company" ;
    Apple_Inc_PN = acePN "Apple_Inc" ;
    Beats_PN = acePN "Beats" ;
    Silicon_Grail_Corp_Chalice_PN = acePN "Silicon_Grail_Corp_Chalice" ;
    Silicon_Grail_PN = acePN "Silicon_Grail" ;
    acquire_V2 = aceV2 "acquire" "acquires" "acquire";
```

Finally, line 20 uses ACE-in-GF to compile the Attempto grammar and the vocabulary extracted from the ontology into a portable grammar format.

4.3 Sentence Construction

Given an input *atom*, the generated **pgf** file and a mapping file, the sentence generator implements the algorithm presented in Procedure 2. Lines 1-3 initialize variables as well as load the information in the **pgf** and mapping files. Line 4 finds the atom in the mapping file that has the same name as the input *atom*. We call it *model_atom*. Line 5 finds the *syntax_tree* of the *model_atom*. The for-loop in lines 6-12 replaces parts of the syntax tree with the mapping of arguments of *atom*. This process creates a new syntax tree which keeps the same structure as the *model_atom*'s syntax tree. Finally, line 13 converts the new syntax tree back to a sentence.
As an example, given an annotated ontology describing Apple Inc. and its acquired companies as mentioned in Section 4, from the set of atoms:

```
acquire(Apple,Sg).   own(Sg,Sgcc).   acquire(Apple,Sgcc).
```

we are able to generate the following sentences:

```
Apple_Inc acquires Silicon_Grail .
Silicon_Grail_Corp_Chalice is a company of Silicon_Grail .
Apple_Inc acquires Silicon_Grail_Corp_Chalice .
```

The above example illustrates the feature of `nlgOntology`^{A}; it emulates the annotations in the ontology to generate sentences. From the annotations provided for the specific case *"Apple acquires Beats",* `nlgOntology`^{A} can generate sentences for other cases that have similar meaninng but with different objects. The repetition of narration

Algorithm 2 Sentence re-construction

Require: an atom
Require: portable grammar format file, mapping file
1: $a \leftarrow$atom
2: $pgf \leftarrow$load pgf
3: $map \leftarrow$mapping
4: $model\_atom \leftarrow map.keys.find(name(a))$
5: $syntax\_tree \leftarrow map(model\_atom)$
6: **for** *part* is a part of $syntax\_tree$ **do**
7: **for** arg, arg_index in $arguments(model\_atom)$ **do**
8: **if** $part == map(arg)$ **then**
9: $syntax\_tree[part] = map(arguments(a)[arg\_index])$
10: **end if**
11: **end for**
12: **end for**
13: $sentence = pgf(syntax\_tree)$

can be seen in many question-answering systems. In particular, `nlgOntology`^{A} uses the annotation of `acquire(Apple, Beats)` to generate the sentences for `acquire(Apple, Sg)` and `acquire(Apple, Sgcc)`. The sentence generation for `own(Sg,Sgcc)` provides the annotation for `own(Apple, Beats)`.

5 Related Work and Analysis

The closest effort to what proposed here is the work in [3], which reports on generating natural language text from class diagrams. In [3], the author developed a system to generate specifications for UML class design while the present work focuses on natural language text generation for a given ontology and a Grammatical Framework, which is manually encoded or automatically generated from the annotations of the ontology.

The work in [11] targets generating an ASP program from controlled natural language, and vice versa. The author uses a bi-directional grammar as the intermediate conversion in combination with reordering atoms for aggregation. There is a correlation between our work and the work in [11] in terms of processing the controlled input format and generating the natural language text. The key difference between our work and that work in [11] is that our system only relies on the structure of the annotated sentences (for `nlgOntology`^{A}) in the text generation and thus could potentially be more flexible.

In order to assess the feasibility of our approach to automatically generate text based on an ontology, we performed an experiment using the *Software ontology*,[8] which is apart of The Open Biological and Biomedical Ontology (OBO) Foundry.[9] We annotated some concepts in the ontology using the tag `oboInOwl:comment` as in the following example:

```
<!-- http://edamontology.org/operation_0244 -->
<owl:Class rdf:about="http://edamontology.org/operation_0244">
  <rdfs:subClassOf rdf:resource="http://edamontology.org/operation_0243
      "/>
```

[8] http://theswo.sourceforge.net
[9] http://www.obofoundry.org

```
  ...
  <rdfs:label>
    Protein flexibility and motion analysis
  </rdfs:label>
  <oboInOwl:comment rdf:datatype="http://www.w3.org/2001/XMLSchemastring
      ">
    %% @n: protein_flexibility_and_motion_analysis
  </oboInOwl:comment>
  <oboInOwl:comment rdf:datatype="http://www.w3.org/2001/XMLSchemastring
      ">
    %% @lin: a protein_flexibility_and_motion_analysis is a
        molecular_dynamics_simulation .
  </oboInOwl:comment>
</owl:Class>
```

We implicitly bind the annotations with the relation `subclassOf` due to the simplicity of the Software ontology. Given the annotated Software ontology, `nlgOntology`^{A} is able to generate some sentences like

```
A DNA_substitution_modelling is a Modelling_and_simulation_operation .
A Molecular_dynamics_simulation is a Modelling_and_simulation_operation .
A Protein_flexibility_and_motion_analysis is a Modelling_and_simulation .
```

6 Conclusions, Discussions, and Future Work

In this paper, we presented two NLG systems, `nlgPhylogeny` and `nlgOntology`^{A}, for automatic generation of English descriptions for a set of atoms derived from ontologies. Both achieve the goal by creating a GF program and relying on the ability to generate sentences of the Grammatical Framework. `nlgPhylogeny` uses pre-defined resources (e.g., linearizations, vocabularies, etc.) to build the sentence generator (GF program), while `nlgOntology`^{A} extracts and manipulates information directly from an annotated ontology. Observe that the structure of the generated text in `nlgPhylogeny` is richer than that in the current `nlgOntology`^{A} due to the fact that the pre-defined resources are hand-crafted and `nlgOntology`^{A} employs a very simple grammar for its sentence structure. For this reason, `nlgPhylogeny` can generate sentences that are more complex than the sentences generated by `nlgOntology`^{A}. On the other hand, `nlgOntology`^{A} relies on meta-information in the ontologies and can be used in any ontology that is annotated and can be parsed by an Attempto Controlled English parser. As such, `nlgOntology`^{A} can save significant efforts before it can be deployed in an application.

We conclude the paper with a short discussion about the applications and possible extensions of `nlgOntology`^{A}. It is easy to see that the current system can be very useful in applications that require shallow explanations. We envision the possibility of using `nlgOntology`^{A} for query-answering or information retrieval systems that, at the end of their complex computations, need to present the result—a set of atoms—to their users and do not need to explain the computation process. In such systems, the answers are often crafted manually or using some templates. This is certainly achievable with `nlgOntology`^{A} as such templates can be provided as annotations for instances in the ontologies. `nlgOntology`^{A} can add some flexibility to such system if multiple linearizations for an instance are provided in the ontologies, since they are translated to potentially different syntax trees. This will result in different sentences

during the generation phase. The current system is, on the other hand, not as good, compared to `nlgPhylogeny`, in dealing with ordered sets of atoms, i.e., the explanation needs to be presented in a certain order. For example, `nlgPhylogeny` needs to present a plan which is a set of atoms with an ordering in the second parameter of the atoms $occur\_concrete/2$ to the users. The implementation of this feature in $\texttt{nlgOntology}^A$ will be our main immediate future work. This will allow $\texttt{nlgOntology}^A$ to provide natural language explanation detailing the steps involved in the computation of a result (e.g., the steps of a procedure or workflow).

To improve the usability of $\texttt{nlgOntology}^A$, we intend to extend the system to consider the problem when the ontology comes with annotations in natural language, i.e., to remove the restrictions that the ontology is annotated using controlled natural language. Interestingly, this idea is closely related to the idea proposed in a Blue Sky Ideas of the 17th International Semantic Web Conference [12].

Acknowledgement

We thank the reviewers for the comments and the references, especially [12]. We would like to acknowledge the partial support of the NSF grants 1458595, 1401639, and 1345232.

References

1. A. Stoltzfus et al.: Phylotastic! Making tree-of-life knowledge accessible, reusable and convenient. BMC Bioinformatics **14** (2013)
2. Angelov, K., Bringert, B., Ranta, A.: Pgf: A portable run-time format for type-theoretical grammars. Journal of Logic, Language and Information **19**, 201–228 (2010)
3. Burden, H., Heldal, R.: Natural language generation from class diagrams. In: Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa 2011), Wellington, New Zealand, ACM (2011)
4. Fuchs, N.E., Schwitter, R.: Attempto controlled english (ace). CoRR **cmp-lg/9603003** (1996)
5. Kamp, H., Reyle, U.: From discourse to logic. Springer (1993)
6. Nguyen, T.H., Son, T.C., Pontelli, E.: Automatic web services composition for phylotastic. In: Practical Aspects of Declarative Languages - 20th International Symposium. pp. 186–202 (2018)
7. Prosdocimi, F., Chisham, B., Thompson, J., Pontelli, E., Stoltzfus, A.: Initial implementation of a comparative data analysis ontology. Evolutionary Bioinformatics **5**, 47–66 (2009)
8. Ranta, A.: Grammatical framework. Journal of Functional Programming **14**(2), 145–189 (2004)
9. Ranta, A.: Grammatical framework: Programming with multilingual grammars. CSLI Publications, Center for the Study of Language and Information (2011)
10. Reiter, E., Dale, R.: Building natural language generation systems. Cambridge university press (2000)
11. Schwitter, R.: Specifying and verbalising answer set programs in controlled natural language. arXiv preprint arXiv:1804.10765 (2018)
12. Vrandečić, D.: Capturing meaning: Toward an abstract wikipedia, http://ceur-ws.org/Vol-2180/