HIT-MSRA Summer School 2012-07-16

Practical Project: Natural Language Generation

Albert Gatt
Institute of Linguistics, University of Malta
Tilburg center for Cognition and Comunication (TiCC)
Department of Computing Science, University of Aberdeen

1. Overview

This project focuses on building a small NLG data-to-text application. The domain you will be working with is that of Neonatal Intensive Care; this is the domain in which the BabyTalk systems were designed.

The aim of this project is to get you to work with some real data, and try to convert it into text by going through the stages of:

1. Document Planning (mainly to structure the messages in the text into a document plan tree)
2. Microplanning (mainly to work out the temporal properties of the events mentioned in the text)
3. Realisation (using an existing realisation engine called simpleNLG to map from the outcome of microplanning to actual English).

1.1 A note for programmers

This practical assumes familiarity with Java, and an ability to work with Java APIs. If you're more comfortable with another programming language, feel free to use it. However, this will mean that you'll be unable to use some of the libraries and APIs provided (described below) and will have to work out your own solution to replace them.

Whichever language you use, I strongly recommend an object-oriented approach to the tasks here.

1.2 A note for non-programmers

If you're not comfortable programming, you can still do most of this practical. Just focus on the parts which do not have "for programmers" in the title.

2. Resources required

Apart from Java, you will need to have installed the following:

1. An IDE (such as Eclipse or NetBeans which supports Java)
2. **Protégé** – this is an editor that allows you to browse ontologies encoded in the OWL description logic. Protégé is freely distributed by Stanford University. It can be downloaded from: http://protege.stanford.edu/

The following resources have been provided:

3. **project.zip** – this is a skeleton NLG system, which provides the basic APIs, to facilitate the implementation. It is subdivided into:
    a. **src/** -- the actual java code directory (this should be easy to import into an IDE like eclipse)
    b. **doc/** -- javadocs for the code provided
    c. **lib/** -- contains other jar files which need to be in your classpath. Note in particular the presence of:
        - **Carados** – a library for reading in and working with OWL ontologies. Note that this requires two further java libraries to work: **aries** and **owlapi-bin**.
        - **SimpleNLG** – a Java API that greatly simplifies the process of generating text in English. SimpleNLG is fully documented and downloadable from this page: http://code.google.com/p/simplenlg/
4. **data/** – this directory contains the input data, as well as some example output texts and document plans. This is further described in Section 3 below.

Note: Programmers should spend some time familiarising themselves with the application, the APIs and the JavaDoc.

3. Outline of the data provided

The BabyTalk systems take raw data (in the form of numbers, symbols etc) as their starting point, and the stages of signal analysis and data interpretation map this input onto an ontology, creating instances of various events.

For this project, you will not be doing any signal or data analysis/interpretation. You have been provided with the ontology, with the instances already set up.

You have the following at your disposal in the directory called ***data/*.**

a. **babytalk.owl** is the ontology developed for BabyTalk. The ontology represents the entities and events in the domain (as well as possible relations between them).

b. **scenario.owl** is the same ontology as **babytalk.owl**. However, it also contains all the instances that were created from the raw data related to a real hospital patient, collected over a 12-hour shift in the NICU.

c. **scenario_computer.html** is the text that the BT-Nurse system generated automatically from the data. Only one section of the document is shown, namely the one related to the patient's respiration.

d. **scenario_docplan.txt** is a text file containing the actual document plan produced by the Bt-Nurse system for this data, for the part of the document shown in scenario_computer.html. The document plan is the basis for the generation of the text in scenario_computer.html. Note that the document plan is a tree, and its leaf nodes are actually instances in the ontology (which is what BabyTalk uses as its "messages").

e. **scenario_docplan_simplified.txt** is a simplified version of the document plan, where only one section of the document is shown, namely the section related to Events During the Shift, under Respiratory Support. This will be our focus for this practical.

4. Familiarisation with the domain

In Protégé, open the **babytalk.owl** ontology. Spend a bit of time looking at the concepts represented and especially the properties that each class has.

4.1 Reverse engineering the text

Now, look at the computer-generated text, **scenario_computer.html**. Just focus on the section entitled **Respiratory Support**, and especially on the part titled **Events during the shift**. This is the section that is related to data about a patient's breathing. To familiarise yourself with the domain, you're going to try to 'reverse engineer' the text, by identifying which of the concepts in the ontology have been mentioned in the text. These are basically the messages that the document planner included.

Here are some questions to guide you in this process:

Q1. Read the text on its own first, without reference to the ontology. Try to identify and make a list of the **main events** mentioned in the section. (Hint: focus especially on the noun phrases and the verbs.)

Q2. Now compare your list to the ontology and try to find the actual concepts (or classes) that you think should correspond to the events you identified in your list.

4.2 Analysing a document plan

Q3. Now, open **scenario_docplan_simplified.txt** and **scenario.owl**. Recall that the document plan has leaves corresponding to messages, which are ontology instances; these instances are all to be found in **scenario.owl**. The edges of the document plan tree represent relations between the instances. Here is an example:

```
+++TSEQUENCE/PARAGRAPH: et_suction_230
++++INCLUDES/PHRASE: purulent_secretion_227
++++INCLUDES/PHRASE: mucoid_secretion_226
```

Figure 1: Document plan snippet

The above specifies a section of the document with the following features:

- This part of the document is a paragraph. It consists of a temporal sequence (TSEQUENCE: a series of events in time), the main one of which is an ET_SUCTION.
- This paragraph has two further children, both phrases, and the plan indicates that they are INCLUDEd in the TSEQUENCE. These two children report observations which are usually made after an ET_SUCTION.

In the rest of the document plan, you'll also notice some states, which are instances of a class called NUMERIC_STATE. These summarise some numerical values related to some property of the patient. In this case, they are parameters related to various levels of chemicals in the patient's blood.

You will also see a TREND which, on inspection, turns out to be related to oxygen saturation (the level of oxygen in a patient's blood, or SATURATION_O2). This is an example of an **abstraction**

created during data interpretation, whereby large numbers of numerical readings are grouped under a single class to describe a state of the patient. Without this, we would end up with thousands of values, which are impossible to report and would be very difficult to understand.

Q4. In the ontology, find the instances which are mentioned in the simplified document plan. Look carefully at the **properties** of each of these instances. Some of these properties are actually described in the text, while others are not. Which of the properties of each class are actually mentioned by the text?

Q5. Now look again at the document plan and focus especially on the **relations** between the messages/instances. Note that:

- Some of these relations are actually identified during the data interpretation, and are in the ontology itself. You can find them by looking under the RELATION node in the ontology. For example, there is an instance of a CONTAINS_LINK which relates two instances, an INTUBATION and a DRUG_ADMINISTRATION. This reflects the fact that an INTUBATION usually involves giving the patient this kind of drug.
- Some other relations (e.g. the ELABORATION in the above example) are created by the document planner itself.

5. Document Planning

You will now attempt to create a document plan yourself. As before, we will only focus on **Respiratory Support**, rather than on all body systems. The aim here is to:

1. Select the right ontology instances for the document plan – for this, you should mainly consider using the **importance** property of each instance. Recall from the lectures that importance is computed during data interpretation, and reflects the clinical significance of an instance.
2. Structure these instances into a plan of the text.

5.1 Getting started: Implementing a document plan (for programmers)

For this part of the practical, you will focus on implementing the DocPlanner interface in the NLGApp. Note that a DocPlan is also provided, but of course, feel free to change or extend the code! Formally, the DocPlan is equivalent to a tree data structure, where leaves are ontology instances, and edges can be labelled with rhetorical relations.

5.2 Writing some rules to relate messages

Based on the example and on your analysis for Q5, create some rules which associate messages together (for example, to indicate that certain messages ELABORATE others, or that certain

IF X is-a ET_SUECTION
AND Y is-a SUCTION_SECRETION_OBSERVATION
AND Y occurs at roughly the same time as X

messages CONTAIN others). Your rules should make reference to the CLASS and its properties. An example corresponding to Figure 1 is shown below in figure 2.

Programmers should implement these rules as part of the DocPlanner.

5.3 Choosing content (for programmers)

In the NLGApp class, write code to read in the **scenario.owl** ontology, so that you have the whole set of instances and classes in memory and can pass it to the document planner. You can use the **carados** API for this. Here is a snippet of code that shows you how:

```
import java.net.URI;
mport uk.ac.abdn.carados.OntologyClassHandler;
import uk.ac.abdn.carados.OntologyException;
import uk.ac.abdn.carados.OntologyHandler;
import uk.ac.abdn.carados.OntologyInstanceHandler;
import uk.ac.abdn.carados.owl2.Owl2CaradosOntology();

//instantiate an ontology handler
OntologyHandler ontologyHandler = new Owl2CaradosOntology();

try {
        ontologyHandler.loadOntology(URI.create(<your onto file>));

} catch( Exception e ) {
        Systeml.err.println(e.getMessage());
        e.printStackTrace();
        System.exit(1);
}
```

Code listing 1: Reading in an OWL ontology

In your DocumentPlanner, implement code that executes the following steps:

1. Find the concepts in the ontology which are related to Respiratory Support, and which you identified in Q3 above. You can do this quite easily with the carados ontology handler. See Code listing 2.

```
//check that a class exists
if(ontologyHandler.containsClass(CLASS_NAME)) {

    //get the actual class/concept
    OntologyClassHandler cls = ontologyHandler.getOntologyClass(CLASS_NAME);

    //iterate through the instances of the class/concept
    for(OntologyInstanceHandler instance: cls.getInstances(false)) {
```

2. Each of the concepts will have multiple instances, but of course, not all of them are important enough to include in the text. Use the **importance** value of each instance to make your choice. In particular, experiment with different cut-offs. For example, try to select only instances whose importance is greater than, say, 75.
3. Structure the instances you select into a document plan tree. Try to do this in two different ways:
    a. **The simple way:** Simply order all messages in the tree by time (earliest first).
    b. **A slightly more complex way**: Find the most important messages/instances first (the **key messages**), identify the messages which are related to them by the rules you designed in Section 5.3, then create a paragraph for each key message and related messages.

6. Microplanning

You will now focus on mapping from messages/instances to structures which are more "linguistic", and which can later be realised as text. The main tasks we're going to focus on are:

1. Lexicalisation, that is, choosing the words (verbs, nouns etc) to express a message
2. Choosing the tense (present, past, past perfect etc) for the message

Programming note: For this part of the practical, you should focus on the LexicalTemplate and Microplanner classes.

6.1 Creating lexicalisation templates

In Section 3 and 4, you identified the concepts which need to be expressed in the relevant section of the text. You also looked at which properties are expressed, and which are not. This information is going to be useful in this part of the exercise.

One way in which this can be done is to create **templates** which map linguistic information. Consider, as an example, the *et_suction* instance in Figure 1. The figure below shows all the properties of this instance:

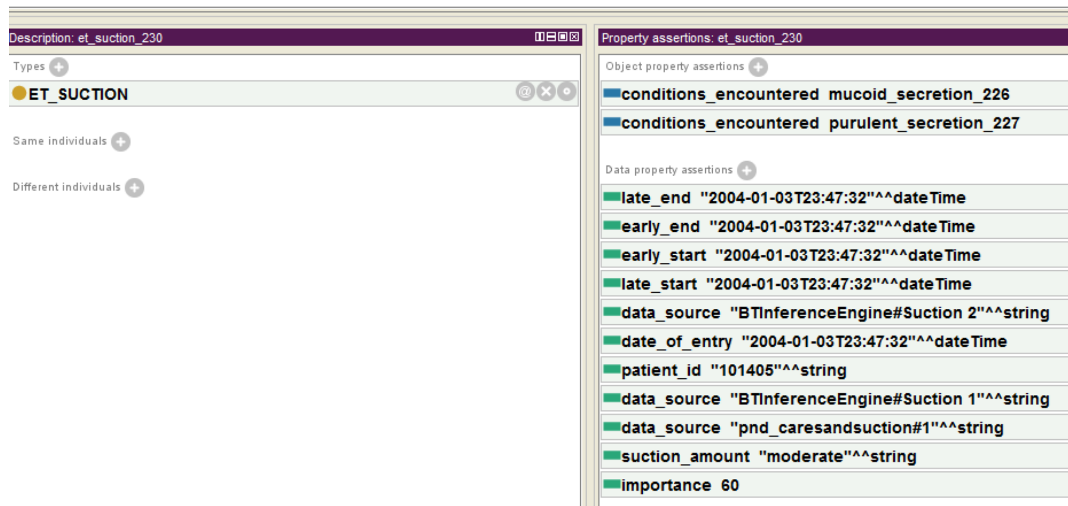

Figure 3: Properties of an instance

The corresponding part of the text is *An ET Suction was done at about 23:45*. Given the instance in Figure 3 and the example text, we could create a template with the information that:

1. the **verb** to use is *do* (since we have *was done* in the text)
2. the **voice** is *passive* (we have *suction was done* not *someone did the suction)*
3. the passive **subject** of the sentence (which is grammatically the object, since it is the thing which was done, not the person who did it) is the name of the class (*ET Suction*).

Go through the small document plan in scenario_docplan_simplified.txt and create a template for all the ontology classes mentioned in this part of the text. Focus especially on which properties of an instance are mentioned, and on how they are mentioned.

Programmers should implement these templates (e.g. by extending the LexicalTemplate class) for use by the Microplanner class.

6.2 Choosing the right tense

Based on what we discussed in the lecture concerning time, we will now look at ways of identifying the right tense. Remember that you need:

1. **Utterance Time**: this can simply be the current time at which you're generating the text;
2. **Event Time**: events in the data have two start times and two end times, which represent the earliest and latest possible start and end. This is because the data is very noisy, so start and end times are represented as intervals. However, you only need to focus on the latest possible start and end. Work out the event times from these.
3. **Reference Time**: you should obtain this from the context. Use the previously mentioned event to get the reference time. If there is no previously mentioned event, then the reference time and the event time are the same. Experiment with different ways of doing

this. For example, what happens if you restrict the context to the current paragraph (the paragraph in which an event is mentioned) when working out the reference time?

Based on these temporal parameters, work out the tense of the event, and add it to the template.

Programmers should consider using the simplenlg TENSE enumerated type for this; that will also make it easier to realise sentences.

6.3 Time modifiers (optional)

In the text, you'll also see that the first event in each paragraph has a time modifier (e.g. *at about 23:45*). Experiment with different ways of adding time modifiers. As an example, here are some alternative ways of saying roughly the same thing:

- An ET Suction was done *at about 23:45*
- An ET Suction was done *late at night*
- An ET Suction was done *in the middle of the night*
- An ET Suction was done *at 23:47*

Programmers should consider adding these modifiers to the Lexical Templates they created earlier.

7. Realisation (for programmers)

At this point, we've planned a small document, and created message templates to choose the words to render it into English. In terms of your implementation, you've worked on your DocPlanner and your Microplanner. Now we'll work on the **NLGRealiser** class.

To realise these templates, you can use SimpleNLG, which provides a simple API to realise sentences in English. You'll probably find it useful to **follow some of the tutorial examples** in the SimpleNLG site to get used to how it works.

The code snippet below shows one way (not the only one!) of realising the ET Suction template we discussed above:

```
import simplenlg.realiser.english.Realiser;
import simplenlg.features.*;
import simplenlg.syntac.english.*;
import simplenlg.framework.NLGFactory;
import simplenlg.lexicon.*;

//initialise a phrase factory and a realise
Realiser realiser = new Realiser(); //realiser for strings
```

8. Informal evaluation

Once you've realised your text, compare it with the relevant part of the computer-generated text and the human text. Can you find any missing content or cases where the language is very different? If so, try to think of the reasons why this occurred.