Ontologies for dialog management and trauma treatment

W. Pasman, M. Tielman, W.P. Brinkman

October 17, 2019

Abstract

This document presents the trauma ontologies software that can interview patients with PTSD to help recall of details of traumatic events. It advances research software that was proven effective. The software uses a hierarchy of concepts, the ontology, to steer the interview. The same ontology files from the original tests can be used with this software. Support has been added to allow other filetypes besides owl. Support for different languages is available, and new languages can be added easily. The original strategy can be used, but alternative strategies used to traverse the concept hierarchy can be created as required. The state of the dialog can be saved and restored at any time. Sample ontologies are provided for a ready-to-use system to handle child sexual abuse and various versions for military post-traumatic stress disorder in various conflict areas. A demo GUI is provided to show the functionality.

Contents

1 Introduction

PTSD patients often have fragmented memories of their trauma and are very reluctant to recall them, requiring detailed questions to stimulate memory retrieval. Previously, a system was developed that showed the effectivity of an ontology-based dialog system to help humans to recall their traumatic experiences [3] These ontologies have been tested and shown effective with real subjects ([1] , [2]).

This document describes the technical details of the traumaontologies module. The code is in Java, and supports the Maven build system for easy incorporation into other maven based code. The main functionality of the module is to generate questions to treat traumas. We start with an overview of the main classes (Figure 1).

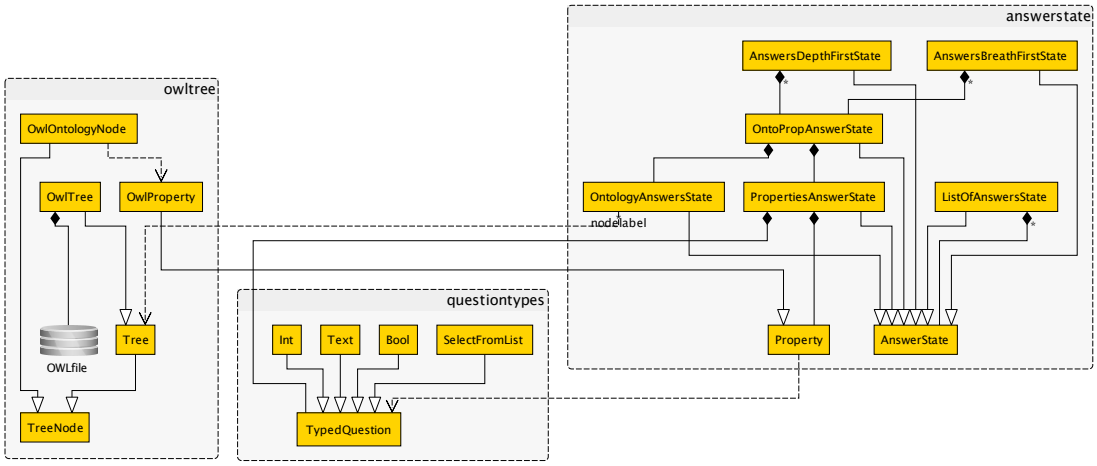

Figure 1: UML diagram of the core code

The question generation process is completely controlled by the AnswerState object. It contains the state of the question-answer process, including all answers received that led up to this state. It has a function `getOptions` to get the next question, and a function `with` that creates a new AnswerState from the answer to the question as given by the user.

The AnswerState in turn depends on its configuration and on a hierarchy, or tree, of relevant concepts.

Regarding the configuration, there are various instances of AnswerState, such as AnswerDepthFirstState and ListOfAnswersState, that result in different interpretation of the concept tree.

The concept tree defines a hierarchy of all concepts that are relevant for the process. The concepts in the tree contain the exact questions plus allowed answers for each concept. The concept tree is currently read from an OWL file, but this can be adapted to support other formats.

The

1.1 example

First an example to illustrate the mechanism. Suppose the tree contains a concept "building" with children "house", "library" and "church", and "house" has children "appartment" and "bungalow". Additionally, we assume that building has properties "size" and "house" has the property "number of inhabitants".

Now suppose that for some treatment, it is useful to ask the user to give details of a relevant building he encountered. We start with creating a OntologyAnswerState pointing at building. This will generate questions (and collect answers) to let the user explain what type of building (jost one building) he is thinking about. So the system may ask "can you clarify the type of building? Is it a house, library or church?". If the user answers "house" the system may ask again "can you clarify if it's an appartment or bungalow?".

Next, it may be required to also ask about the properties of the selected house. To do this we replace the OntologyAnswerState with an OntoPropAnswerState pointing at building. The result will be an interview starting as with the OntologyAnswerState, leading to for instance the user's answer that he has a bungalow in mind. But now the questioning proceeds with "what is the size of the bungalow?" and "what are the number of inhabitants?".

Suppose we want to ask about a special house, say the user's own house. We wrap the OntoPropAnswerState in a AnswerStateExplanationDecorator to add some extra explanation that this question is about the user's own house.

Next, we want to ask about both the user's own house and a second, more general house. We can make two AnswerStates, one with and one without the wrappng Decorator. We put them in a ListOfAnswersState, in the correct order for asking. Notice that we can always re-use AnswerState objects because they are immutable.

If there is an unknown number of relevant buildings, we can put the OntoPropAnswerState into a AnswersDepthFirstState. After the user has answered all there is to ask about one house, he will now get the question if there are more relevant houses.

The following sections detail on the AnswerState and concept hierarchy. The example treatment ontologies are discussed. Also the example GUI is discussed.

2 AnswerState

AnswerState contains the complete state of the dialog. It defines the how the concept tree will be used. It can provide a question to forward the state and it can create a new state given a user's answer to the question.

So it defines a (possibly partial) answer to a question and the answer type needed to bring it closer to a full answer. Specifically, all AnswerStates implement two main functions::

1. getOptions() is to be called to get the question and the allowed answers. The state is complete if this returns null.
2. with(answer) is called to acquire a new AnswerState that includes the given answer

Usually, there will be a loop like this in various places in the code

```
while ( TypedQuestion question=state.getOptions() !=null) {
    present user question.getQuestion()
    repeat {
        answer=get user's answer
    } until question.fits(answer)
    state=state.with(answer)
}
```

2.1 Types of AnswerStates

Different implementations of AnswerState area avilable, all differing in the way the given concept is used. Some of these recursively use other AnswerStates to (recursive) construct a global AnswerState for the entire interview.

1. OntologyAnswerState: This AnswerState visits the children of the initial concept, searching for the best match to what the user has in mind. It offering the user to choose between children nodes repeatedly. The idea is that the initial state is with the most abstract OntologyNode fitting the expectations (typically "object" or "person"). In each node, getOptions() returns a set of more accurate ontology nodes that are children of the current node. By calling with(answer) with one of these nodes as argument, the new state is that selected node. If tnat selected node is a leaf node, the state is final and the process is done. Please refer to the javadoc for all details.
2. PropertiesAnswerState. This AnswerState asks all TypedQuestions (and collects the answers) attributed to the given concept node and all its ancester nodes.
3. OntoPropAnswerState. This AnswerState first executes the OntologyAnswerState until the user has found the best matching concept. Then it proceeds with the PropertiesAnswerState for that concept. So it collects all relevant info for a given root concept.
4. AnswersBreathFirstState. This AnswerState takes a concept and the name of the main property (eg, "Name" which refers to a person's name). The main property must be one of the properties of the given concept. It first loops asking the user if there are more of these items and asks the value for the most relevant property (so, it collect eg a list of person names). Then in a second loop, it executes a OntoPropAnswerState procedure to fill in all the details.
5. AnswersDepthFirstState. This AnswerState takes a start concept (eg "Object" which refers to a physical object that was somehow relevant) It first runs an OntoPropAnswerState procedure on the given concept.. Then it asks if there are more relevant items of this type. If so,it repeats. The result is a list of answers, each detailing on another instance of the concept.

2.1.1 OntoPropAnswerState substitutions

OntoPropAnswerState has a special step when handling getOptions. As mentioned above the state first executes the OntologyAnswerState procedure. The answers from the user are collected in a map with as key the name of the concept and as value Suppose thet the If the user gave an answer "John" to some property, say "Name", then all subsequent `getOption` calls will substitute any occurences of the property name within square brackets, so "[Name]", replace with the exact answer given by the user. So the question "where did you meet [Name]" would be turned into "where did you meet John".

2.1.2 AnswersBreathFirstState substitutions

The AnswersbreathFirstState uses the above mentioned special step substituting a property in the questions. It makes a list of OntoPropAnswerStates, one for each concept (typically, Person) deemed relevant by the user. But instead of going through the full OntoPropAnswerState, only the main property (eg "Name") is filled in in the OntoPropAnswerState. After the user has filled in all main properties (Name), the OntoPropAnswerState will automatically fill in the right names in all remaining questions required to complete the collected OntoPropAnswerStates.

To give an example of how this works, suppose we have a hierarchy of concepts Person with children "father", "mother", "sister" and "brother". All Persons have a property Name and a property Age. An AnswersBreathFirstState procedure would ask repeatedly "Please enter the name of the next relevant person" till all names are collected. Suppose the user mentions "Sue, Peter, Mary, Bob". Then, it would take the first person, and first ask questions like "is Person Sue your father, mother, sister or brother?". After clarifying the exact concept type, the remaining property would be filled in, with a question like "What is [Name]s age" and [Name] can be substituted with Sue because Name is the main property and the answer on the main property question was "Sue". Next, the same procedure is repeated with the remaining names.

2.2 Loading/Saving

The AnswerState can be (de)serialized with Jackson to/from a json formatted object using `objectmapper.writeValueAsString` and `objectmapper.readValue`. Example code is available in PropertiesAnswerStateTest.

2.3 TypedQuestion

TypedQuestion objects contain a question for a human (a text string) and a method `fits` that checks if an answer given by the user matches the expected answers. For example if the question is asking a yes/no question, the `fits` function can check if the answer is yes or no.

This is the complete list of TypedQuestion objects::

1. `Bool`. This is a yes-no question
2. `Int`. This is a question that expects an integer number as answer
3. `Text`. This question expects any free text as an answer.
4. `Word`. This question expects a single word (no whitespace) as answer
5. `SelectFromList`. This expects one from a known list of phrases as answer.

TypedQuestions are generated in two ways:

- indirectly by code in AnswerState, e.g. to probe the user which concepts from the concept tree that he is thinking about.
- directly from the Property objects in the concept tree that contain specific questions to be asked

The TypedQuestion has a function isTranslated(). If this function returns true, answers to this type of question have to be translated. This typically is true for `Bool` and `SelectFromList` questions, but not for `Text` or `Int`,

3 Concept Tree

The concept tree is a tree with each node being a concept that is relevant for the interview. It is a tree with concepts ordered from general to more specific, allowing the interview to start at a high-level concept, with more specific concepts as children. The introduction already mentioned 'building' as a possible general concept with a bungalow as a child instance.

In the code, the nodes are of the concept tree are represented by OntologyNode, which extends a standard AttributeTree. The node has a name (String) and a list of Attributes.

The name strings are used directly to be shown to the user, so it is important to use human-understandable names here.

The Attributes are Property objects that contain a TypedQuestion. These questions are usually open-ended questions, as the system does not interpret answers to questions like "what was the size of the house" other than some general type checking capabilities (eg, if the answer must be an integer number, or yes/no).

3.1 OWL storage format

The collection of attribute trees can be stored in many ways, but this implementation provides an implementation using the OWL format. Other formats can be easily supported as the OWL format is just an instance of the more general interface used by the AnswerStates. The choice for OWL allows the original OWL files from the clinically tested prototype can be used, and the use of the Protege tool for developing the tree.

The OwlTree class converts the database into a Tree, the OwlOntologyNode adapts OwlClass objects to TreeNodes, and the OwlProperty implements the Property interface.

3.2 Class

The `owl:Class` object is used to represent the OntologyNode objects which are the concepts relevant for the interview. A class object in OWL contains an attribute `rdf:ID` and this attribute is used as the name string of the OntologyNode.

3.3 subClassOf

The `rdfs:subClassOf` property is used to define the hierarchy of the classes. It is used to define the parent-children relations between the OntologyNodes.

3.4 Property

The `owl:DatatypeProperty` and `owl:FunctionalProperty` are used to attach properties to a Class. These have a number of attributes that are used in specific ways to create a TypedQuestion objects.

1. `rdfs:label` this attribute contains a full question to be asked to the user during the interview, to elicit the answer
2. `rdfs:comment` behaves the same as `rdfs:label`. Only one of these is allowed.
3. `rdf:ID` this id can be shown to the user as question, if more specific question text is missing
4. `rdfs:domain` the class name (concept) that this property applies to
5. `rdfs:range` the range of answers that is allowed as an answer. Can be `http://www.w3.org/2001/XMLSchema#boolean` (creating a `Bool`), `http://www.w3.org/2001/XMLSchema#int` (creating a `Int`), `http://www.w3.org/2001/XMLSchema#string` (creating a `Text`).

3.5 Included concept trees

The code includes several concept hierarchies that have been used in clinical treatment for a number of PTSDs. The concept hierarchies are contained in the following files and ment for specific PTSD treatments:

- `Child Sexual Abuse.owl`. This is for treating PTSD due to sexual abuse during childhood.
- `War.owl` This is for treating PTSD due to general war situations.
- `War Afghanistan.owl` Treating PTSD due to the Afghanistan war.
- `War Bosnia.owl` Treating PTSD due to the Libyan war.
- `War Libya.owl` Treating PTSD due to the Libyan war.

The example trees have a number of specific concepts that should be handled in the given order

1. `Geografische_locatie`. This concept contains general geographical location information about the event.
2. `Type_Locatie`. This concept contains more details about the type and details of the geographical location
3. `Object`. This concept contains all details about all relevant objects. These should be handled depth-first, asking all information about an object before asking about the next.
4. `Persoon`. This concept contains all details about all relevant persons. This should be handled breath-first: first asking all the names of the relevant persons and then asking more about each person in a second run through all persons.

Therefore the included treatments should be used in combination with the following AnswerState:

```
private void initAnswerState() throws OWLOntologyCreationException {
  OWLOntologyManager manager = OWLManager.createOWLOntologyManager();
  OWLOntology ontology = manager.loadOntologyFromOntologyDocument(
    getClass().getResourceAsStream("/" + treatmentCombo.getSelectedItem()));
  OwlTreeReasoner reas = new OwlTreeReasoner(ontology);
  tree = new OwlTree(reas);

  AnswerState geoanswer = new PropertiesAnswerState("Geografische_locatie");
  AnswerState loctypeanswer = new AnswerStateExplanationDecorator(
    new OntoPropAnswerState(
      new OwlOntologyNode("Type_Locatie", reas), null),"In wat voor soort locatie was U?");
  AnswerState objectsanswer = new AnswerStateExplanationDecorator(
    new AnswersDepthFirstState("Object","Zijn er nog meer relevante objecten?", tree),
    "We verzamelen de relevante objecten op de locatie. We gaan een voor een door alle relevante objecten, b
  AnswerState personsanswer = new AnswerStateExplanationDecorator(
    new AnswersBreathFirstState(
      new OwlOntologyNode("Persoon", reas), "Naam", "Zijn er nog meer relevante personen?"),
        "We verzamelen de relevante personen op de locatie. We verzamelen eerst de namen.");

  dialogState = new ListOfAnswersState(Arrays.asList(geoanswer,
    loctypeanswer, objectsanswer, personsanswer));
}
```

4 Interactive example dialog

Figure 4 shows the example GUI. The example gui uses the treatments that were developed to treat child sexual abuse and war traumas. When the GUI is started up (e.g., double click the `traumaontologies.with-depdencies.jar` file) the user is first prompted to select one of the included treatments. Then, the user gets presented questions. He can type the answer as text in the field above the answer button. When the answer is completed he clicks 'answer'.

The combobox "EN", "NL" allows the user to select a language of his choice. Currently english and dutch are supported but this can easily be extended by creating additional translation CSV files as described in the section 5..

The "Save" shows what would be saved in a pop-up window. The example does not really save data for security reasons.

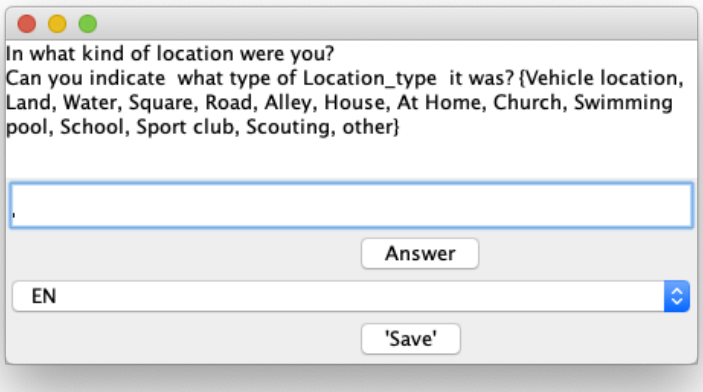

Figure 2: Example GUI

4.1 Connecting with your own GUI

The provided example GUI is completely optional: The GUI is just a simple wrapper of the core traumaontologies module to help the user interact with the code in a simple demo mode. The GUI demo also shows how translation functionality can be easily added on top of this.

Also, the provided treatments are optional. You can use your own files containing a concept hierarchy to drive your own therapy in a similar way.

if you want to use the provided treatments (the `owl` files) you should use the initial answer state as was discussed in section 3.5.

5 Translation

The example code in the traumaontologies module returns sentences in Dutch. This is just a choice, ontologies can just as well be written directly in another language. Only some sentences in the system are hard coded, to change this one would have to download and modify the source code.

We recommend to use the translator module (https://tracinsy.ewi.tudelft.nl/pubtrac/Utilities/wiki/Translator) as used in our example code. The example code shows how to use our translator module to translate the Dutch sentences into English.

Note that the translation files may contain variables of the form [Naam] which may look similar to the variables used in the AnswerStates (section 2.1.1). These should not interfere. All variables in the question sentences should have been replaced by the time they reach the Translation package. The translation package figures out variable substitutions by itself, assuming it receives plain text without any explicit variables.

References

[1] Tielman, M., Neerincx, M., Bidarra, R., Kybartas, B., and Brinkman, W. A therapy system for post-traumatic stress disorder using a virtual agent and virtual storytelling to reconstruct traumatic memories, 2017.

[2] Tielman, M., van Meggelen, M., Neerincx, M. A., and Brinkman, W. P. An ontology-based question system for a virtual coach assisting in trauma recollection., 2015.

[3] Tielman, M. L. *A Virtual Agent for Post-Traumatic Stress Disorder Treatment.* 2018.