UNIVERSITY OF GOTHENBURG

Project: Recipe Dialogue System

Student name: *Dominik Künkele*

Course: *Dialogue Systems 2 (LT2319)*
Due date: *November 1st, 2022*

Todo list

Introduction

In this project,I wanted to create something, I could actually use in my life. Since I like very much to cook, building a dialogue system that supports here, was not far fetched. The goal however was not only to create a system that can handle only few recipes and situations, but can also be extended easily with many more recipes. For this, I wrote a script in *python* that can translate a more user friendly description of a recipe into the necessary goals, actions and so on in TDM.

APIs

I didn't use any explicit external API in the project. However, I relied heavily on the http service to add functionality to the dialogue system and lookup information from local files. This includes firstly a lookup dictionary for detailed information about steps in the recipes. Here, I listed the needed ingredients for each step with their amount and form (e.g. onions: two, roughly chopped), used objects and their temperature (e.g. oven: 180 degrees) as well as information about time end ending conditions of a step. With this service, I was able to feed the system the needed information for a specific step. Was the information not specified in the current step, it utilized the information from to the last step. With this approach, I could respond to retroactive questions of a user. A simple lookup file lookes like the following:

```
"omelette_recipe": {
    "ingredients": {
        "egg": {
            "amount": "three",
            "form": null
        },
        "salt": {
            "amount": "a pinch",
            "form": null
        }
    },
    "steps": {
        "omelette_recipe_step_0": [
            {
                "time": null,
                "condition": null,
                "ingredients": {
                    "egg": {
                        "amount": "three",
```

```
                        "form": null
                    }
                },
                "objects": {
                    "bowl": {
                        "temperature": null
                    }
                }
            }
        ],
        "omelette_recipe_step_1": [
            {
                "time": null,
                "condition": null,
                "ingredients": {
                    "butter": {
                        "amount": null,
                        "form": null
                    }
                },
                "objects": {
                    "pan": {
                        "temperature": "high heat"
                    }
                }
            }
        ]
    }
}
```

Listing 1: Example of recipe loopup

Secondly, I used a simple dictionary to lookup subsitute for ingredients. Unfortunately, I couldn't find an API for it, but just text on a website. To make the http service faster and simpler, I downloaded the data and stored it in a json file, which can be easily accessed by the service.

Data collection

For the data collection I recorded four dialogues. Two of them lasted around 10 minutes, while only the first minute was recorded for the other two. Still, these short beginnings held some interesting information and I distilled and analyzed them as well.

The dialogues were recorded in a roleplay scenario and not actually while cooking. This of course has a high impact on how the participants speak and real dialogues would bring more authenticity and should be included in the future.

Analysis of content. First, I tried to analyze the dialgues in respect to the domain and how the participants were talking about the topic of cooking. The instructions consisted of a verb that was either intransitive, transitive or ditransitive. The transitive and ditransitive verbs only referred to *ingredients* or *objects*. If it was an ingredient, they could add the amount (e.g. 'two teaspoons') to use or the form, it should be in (e.g. 'roughly chopped'). Objects could optionally be enriched with information about it's temperature. In some cases, the instructor could add also information to the whole step about time (e.g. 'for three minutes') or an ending condition (e.g. 'until it appears glassy').

S>: OK, perfect. And also ähh juniper berries.
U>: Hmm. I have.
S>: Hmm..ok. Around a teaspoon.
U>: Hmm

Dialogue 1: Ingredient with amount

S>: And now you can start preheat the oven for 220 degrees.
U>: I have done it. What should we do?

Dialogue 2: Object with temperature

S>: Leave the meat for further 15 minutes.
U>: OK, done.
S>: The rind should be very crusty, but not burned.
U>: Hmm, hmm.
S>: So around 15 minutes. If takes longer, leave it longer. If it is already...
U>: Should I take a look sometimes?

Dialogue 3: Step with time and condition

Analysis of dialogue. Looking at how the dialogue was built up, I could also find some interesting ideas. These can be differentiated into behaviour of the system and behaviour of the user.

User behaviour. It was very noticeable, how often the user was acknowledging and confirming the utterances by the system. I could differentiate three different types. In the first, the user was just using one word (e.g. 'ok', 'yes') or a non-lexical sound (e.g. 'hmm', 'uh-hmm'). In the second one, the user repeated a word from the previous system utterance. This can also be seen as a perceptual confirmation in some cases. In the last, the user describes the current situation and by this verifies it with the system. Dialogues 4 and 5 show some examples.

S>: Then 400 grams of mixed mushrooms.
U>: Mixed mushrooms, OK, OK, Champions, do they work? (Type 2)

Dialogue 4: Acknowledgement (Type 2)

S>: OK, you can try to feel the meat with the tip of a knife. if it's tender or not. Is it tender?
U>: It's bleeding. (Type 3)
S>: Then you can give it another 30 minutes in the oven.
U>: OK. (Type 1)
S>: And the oven goes back up to 220 degrees.
U>: Hmm (Type 1)

Dialogue 5: Acknowledgement (Type 1 and 3)

- user doing steps earlier - explicit substitutions, without proposal –> memory - jokes - ask actions

System behaviour. - separate ingredient/amount - system was waiting for user to request step - adapt language to user

Implementation

The goal of my project was to create a dialogue system that can handle user input as flexible as possible, but at the same time is also easy to extend. This especially applied to adding new recipes to the system. For this, it is now possible to write recipe instructions in a rather simple XML file. A *generator* is translating this file into the necessary changes of the TDM files (Ontology, Domain, Grammar, nlg, expected input). There are two main reasons for this approach.

First of all, an end-user shouldn't need to have too much knowledge of TDM to adapt the system to his needs. Adding a recipe involves changing a lot of different files. All these changes furthermore need to be aligned and are not allowed to be in conflict with each other. If a user would only need to change one file with a rather simple syntax, the user experience would be much better, and errors will be prevented.

The second reason that was also much more important for myself in the last weeks is the simplification of the develeopment process itself. With this approach, I didn't need to worry of forgetting to change files or to add specific tags to a file. I only needed to define them once in the *generator* and it was applied to all recipes. Furthermore, this made it much easier to change a certain behaviour that applied to all recipes, since I only needed to change the template instead of every single recipe itself. And lastly, this also gave me a quick lookup, how I implemented a feature and which files it involved.

One restriction of this approach is that I could implement features only in a generic way, since it needs to take all recipes into account. Very specific recipe-dependent requirements couldn't be implemented in an easy way. On the other hand, this makes the system predictable and the user a better understanding of the features and when to use them.

Generator. The generator is adapting the content of the following files: *domain.xml*, *ontology.xml*, *grammer_eng.xml*, *nlg.json* and *expected_input.json*. Some parts of these files are recipe-independent. These static parts can be added to the corresponding template file. The generator will then read the XML file containing the recipes (*recipes.xml*) and insert the dynamic recipe-dependent parts into the templates. Furthermore, it will generate a *recipe lookup* file that contains information about each step of all recipes.

recipes.xml. The *recipes.xml* file has the following structure:

```
<recipes>
    <recipe name="omelette" image_url="www.link.to/image">
        <utterances>
            <utterance>make an omelette</utterance>
            ...
        </utterances>
        <ingredients>
            <!-- can have attributes 'amount' and 'form' -->
            <ingredient name="onion" amount="two" form="roughly chopped">
                We need two onions
            </ingredient>
            ...
        </ingredients>
        <steps>
            <step>
                <!--
                    - can have general attributes 'time' and 'condition'
```

```
                    – can have attribute 'ingredient' with optionally 'amount' and 'form'
                    – can have attribute 'object' with optionally 'temperature'
                --->
                <substep ingredient="egg" amount="three" object="bowl">
                    Crack three eggs on a bowl
                </substep>
                <substep ingredient="salt" amount="a pinch">
                    whisk with a pinch of salt.
                </substep>

                <!-- optional -->
                <how>
                    <step>
                        <!-- substeps can't contain any attributes -->
                        <substep>Strip the leaves off with you fingers</substep>
                        ...
                    </step>
                    ...
                </how>
            </step>
            ...
        </steps>

        <finisher>Now serve and enjoy your meal</finisher>
    </recipe>
    ...
</recipes>
```

Listing 2: Example of recipes.xml

In the following paragraphs, I will include the corresponding tag with an emphasized attribute in square brackets. If for example the *domain.xml* uses the *name* attribute of the *ingredient* tag, this will be referenced as [<ingredient *name*="..." />].

Domain. Every recipe will generate a *perform goal* in the *domain.xml*. Each *perfom goal* consists of two parts, the listing of the ingredients and the step-by-step instructions. The analysis of the recorded dialogue showed that the user doesn't necessarily acknoledge every ingredient, but just let's the system/instructor read all ingredients. That's why, I included a way to optionally acknowledge an ingredient. If the user doesn't say anything, the system automatically goes on with the next ingredient after a pause of three seconds. The following code is added for each ingredient:

```
<if>
    <proposition predicate="omelette_recipe_salt_read" value="false"/>
    <then>
        <inform insist="true">
            <proposition predicate="read_ingredient_list" value="omelette_recipe_salt"/>
        </inform>
        <end_turn expected_passivity="3.0"/>

        <forget predicate="proposed_ingredient"/>

        <assume_shared>
            <proposition predicate="omelette_recipe_salt_read" value="true"/>
        </assume_shared>
        <assume_shared>
            <proposition predicate="which_ingredient" value="salt"/>
        </assume_shared>
```

```
    </then>
</if>
```

Listing 3: Listing of ingredients

The reason for this complicated structure is the behaviour if the <*end_turn*/> tag. When the user only acknowledged the ingredient or didn't say anything, the system worked perfectly fine. But as soon as the system switched to another goal (after the user e.g. asked a question) and returned back, it got stuck in the *expected_passivity*. For this, I introduced a *boolean* variable that kept track if an ingredient was already read to the user. If so, it would be skipped, when returning back to this goal.

Furthermore, I stored the name of the ingredient [<ingredient *name*="..." />] in the predicate *which_ingredient*. This will get useful later, when the user raises questions like "How much?" or similar without explicitly specifying the ingredient, they are talking about.

The predicate *proposed_ingredient* will be exlained later

explain proposed ingredient

Each step of a recipe will add the following to the *goal* in the domain:

```
<assume_shared>
    <proposition predicate="current_step" value="omelette_recipe_step_0"/>
</assume_shared>
<assume_shared>
    <proposition predicate="which_ingredient" value="salt"/>
</assume_shared>
<assume_shared>
    <proposition predicate="which_object" value="bowl"/>
</assume_shared>
<get_done action="omelette_recipe_step_0"/>
```

Listing 4: Step of a recipe

The analysis showed that the user always acknowledged a step in a recipe. For this reason, I used a *get_done* element. As before, I try to give the system as much information as possible about the current state. Accordingly, I fill the predicates *current_step*, *which_ingredient* and *which_object* [last substep in step: <substep *ingredient*="..." *object*="...">].

If the step contains a *how* element in the *recipes.xml*, the generator will create also a new *perform goal* with the name of the step. This new goal will only contain the *get_done* elements and will not share any propositions. The reason for this is that the effort for implementing it seemed complex while the benefit was not to big. In the recorded dialogues, the users never asked further questions in these situations. In the future, this could be implemented to also react to the less common responses.

Sample dialogues

Discussion

Future work