University of Camerino

School of Science and Technology

Master degree in Computer Science (LM-18)
Knowledge Engineering and Business Intelligence

Intelligent Meal Suggestion System for Restaurants Using Knowledge-Based Solutions and Ontology-Driven Meta-Modelling

Authors:
Chiocchi Francesco
Rossini Nicolò

Supervisors:
Prof. Knut Hinkelmann
Prof. Emanuele Laurenzi

A.Y. 2023/2024

Contents

Listings

List of Figures

1. Introduction

This document aims to outline the process undertaken to complete the Knowledge Engineering and Business Intelligence project. Section 1.1 will provide a description of the system to be developed, while Section 1.2 will define the tasks to be accomplished. Following this, we will examine the System Design phase (Chapter 2), the implemented Knowledge-Based Solutions (Chapter 3), and the description of our implementation of Agile and Ontology-based Meta-Modelling.
In the Conclusion chapter (Chapter 5), we will offer our personal perspectives on the solutions presented.

The solution developed for Task 1 can be found at this link:
https://github.com/FrancescoChiocchi8/KEBI-Project, while the solution for Task 2 can be found at this link: https://github.com/FrancescoChiocchi8/KEBI-Project

1.1 Project Description

Many restaurants have their menus digitized. Guests can scan a QR code and have the menu presented on their smartphones. A disadvantage is that the screen is very small and it is difficult to get an overview, in particular, if the menu is large. However, some guests can not or do not want every meal, e.g. vegetarians or guests with an allergy. Instead of showing all the meals that are offered, it would be preferable to show only those meals the guest prefers. The objective of the project is to represent the knowledge about meals and guest preferences and create a system that allows to select those meals that fit the guest preferences. The knowledge base shall contain information about typical meals of an Italian restaurant, e.g. pizza, pasta, and main dishes. Meals consist of ingredients. There are different types of ingredients like meat, vegetables, fruits, or dairy. For each ingredient, there is information about the calories. Guests can be carnivores, vegetarians, calorie-conscious, or suffer from allergies, e.g. lactose or gluten intolerance.

1.2 Project Tasks

The following are the tasks to tackle for completing the Knowledge Engineering and Business Intelligence projects:

1. Create different knowledge-based solutions for recommending food depending on the profile of a guest (carnivores, vegetarians, calorie-conscious, suffering from allergiesetc.) usind the following representation languages:
    - Decision tables (including DRD with sub-decisions and corresponding decision tables);

- Prolog (including facts and rules);
- Knowledge graph/Ontology (including rules in SWRL, queries in SPARQL and SHACL shapes)

2. Agile and ontology-based meta-modelling: adapt BPMN 2.0 to suggest the meals for a given customer. For this, you can re-use or extend the knowledge graph/ontology created in the previous task. One option that you have is to specify the class BPMN Task with a new class and add additional properties, similar to what we have done in class with the Business Process as a Service case. Think of a new graphical notation for the new modelling element, which could be easy to understand for the restaurant manager. Use the triple store interface (Jena Fuseki) to fire the query result.

2. System Design

In this chapter, we will present our concepts regarding the data inputs and outputs that the system will analyze.

2.1 Input format

We have designed an input form, thanks to Google Modules and Camunda (for the output view), that will serve as the foundation for the rest of this project. An image illustrating our concept of the input form is shown below (Figure 2.1). This form is intended to be completed by the customer to generate a list of the best meals based on their preferences.

In this form, presented in Figure 2.1, we can see that the first choice to be made is to select either "Carnivorous" or "Vegetarian" (represented by circles, indicating that only one option can be selected). This choice is not mandatory (as indicated by the absence of an asterisk in the figure), meaning that a user may choose not to identify as either Carnivorous or Vegetarian. If this field is left blank (equivalent to an empty string, implying that the user identifies as omnivorous), the list of meals considered will not be filtered and will remain unchanged.
Regarding one of the project objectives—to show users only a portion of the menu based on their preferences—a thorough analysis was conducted. Consequently, we concluded that the complete list should still be available to avoid limiting the user (e.g., if the user wishes to view the entire list of meals, this option must be provided).
Further down, there is another choice where the user selects their interest level in consuming high-calorie foods (0-level), medium-high calorie foods (1-level), medium-low calorie foods (2-level), and low-calorie foods (3-level). This choice is mandatory, and by default, level 0 is selected.
Furthermore, the user is asked to specify any allergies they might have, represented in the figure by rectangles that allow multiple selections (e.g., both Lactose and Gluten can be selected). If the user does not specify any allergies, it is assumed they have none, and, as with the Carnivorous and Vegetarian options, the entire menu list will be considered.
Finally, the user can complete the corresponding section pertaining to the course. For example, they may enter "Appetizer," "First Course," "Main Course," "Pizza," "Drink," or "Dessert."
After this food filtering process, the system can also provide information about the meals offered by the Italian Restaurant, including the ingredients and calories of each dish, through an optional field below the allergies section.
Lastly, in the final field of the figure, users can view the calorie content of each individual portion of the meal.

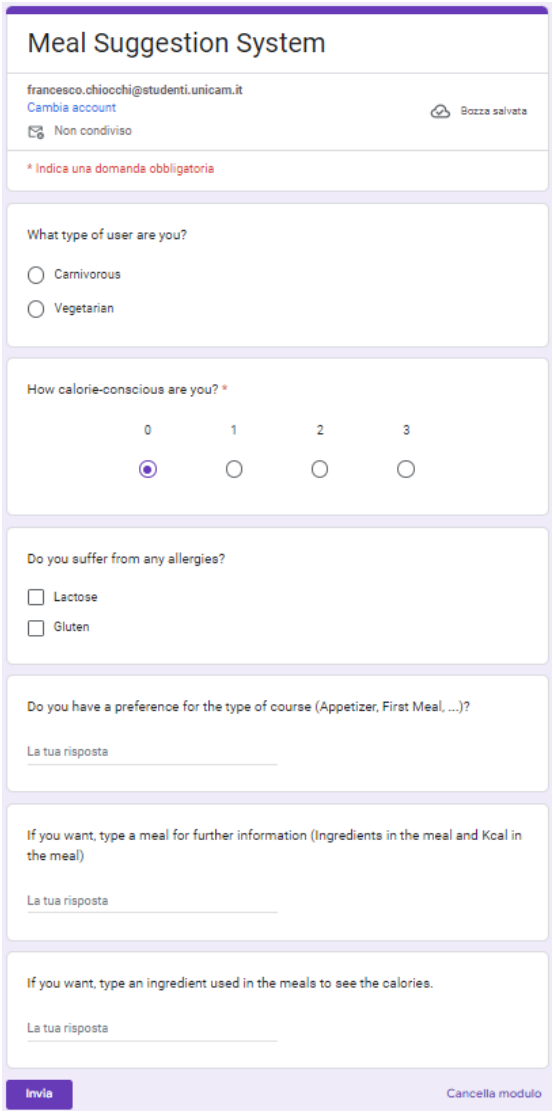

Figure 2.1: Sample Input form for the customer

2.2 Output format

In the Figure 2.2 we can see an example of how the form might be completed by the user.

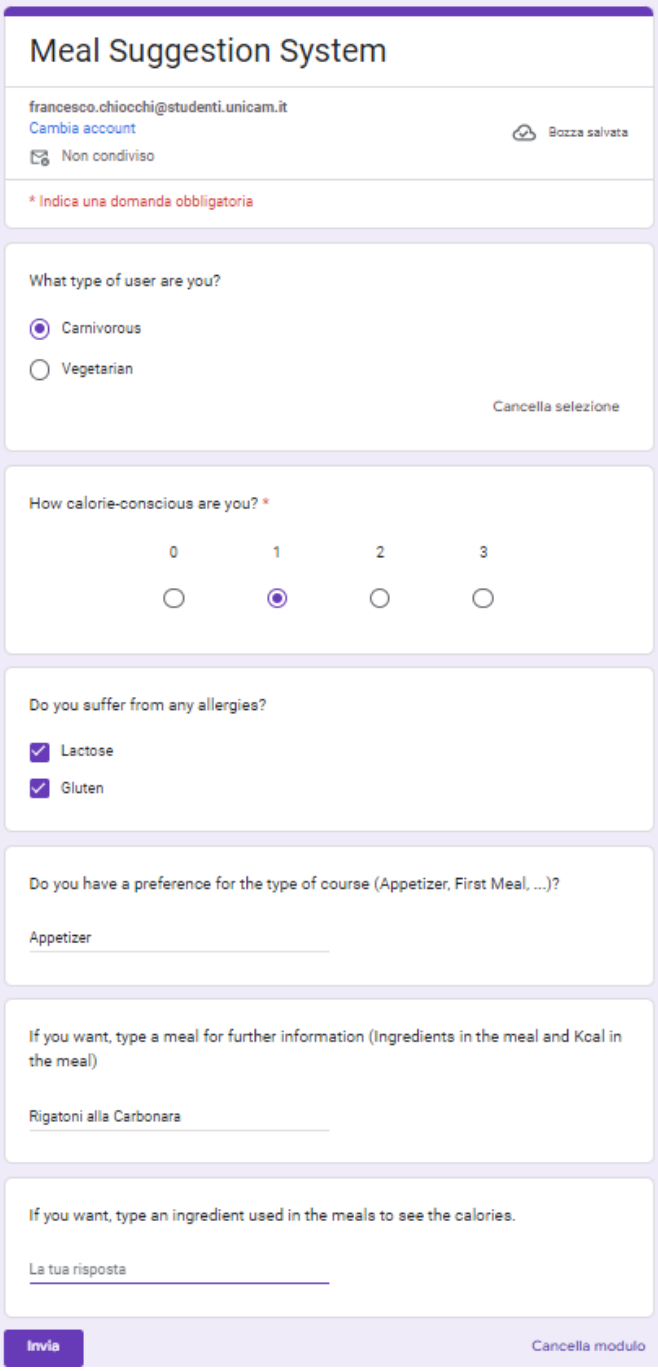

Figure 2.2: Sample filled out form

Suggested Meals
Affettati Misti

Table 2.1: Output Preference for the Figure 2.2

Ingredient in a Meal
Salt
Oil
Sheep's cheese
Egg
Pasta
Black pepper
Pork cheek

Table 2.2: Return the Ingredients in a meal; in this case with reference to 2.2 the meal is "Rigatoni alla Carbonara"

3. Knowledge Based Solution

Relying on the input form and format previously defined, we can query our knowledge base in three different ways. Firstly, by using Decision Tables (Section 3.1) to match user preferences and filter out undesired meal. With Prolog (Section 3.2), we can extend the logic provided by the Decision Tables by applying the functionalities provided by a programming language. This allows the users to manipulate data and perform even more actions. Finally, using Knowledge Graphs (Section 3.3, we can store our data in a convenient format and query them as if they were stored in a database. Following, in this chapter, we will discuss each of the three solutions, their main advantages and drawbacks.

3.1 Decision Model

In the realm of business analysis, the Decision Model and Notation (DMN), as outlined by the Object Management Group (OMG), stands as a recognized standard. This methodology finds application in delineating and structuring recurring decisions within organizations, facilitating a format where decision models can be seamlessly shared across entities. By adhering to this standard, companies can establish a unified modeling notation, empowering them to make decisions conducive to effective decision management and adherence to business rules.

DMN Elements

The DMN standard comprises four key elements:

- Decision Requirements Diagrams: These diagrams elucidate the interconnections among decision-making components, forming a dependency network.
- Decision tables: These tables offer a succinct visual depiction for specifying actions contingent upon provided conditions.
- Business Context: This pertains to the contextual factors influencing the decisions within the company.
- Friendly Enough Expression Language (FEEL): FEEL serves as a language utilized for evaluating expressions within decision tables.

3.1.1 Camunda

Camunda [1]is an open-source platform, built in Java, specifically engineered to facilitate workflow and process automation, catering to companies of varying scales. Its functionalities extend to the creation of decision models and, more broadly, BPMN process

[1]Camunda: https://camunda.com/

diagrams and DMN decision tables. Additionally, Camunda offers [2] REST APIs, enabling developers who do not utilize Java to access its features. We leveraged Camunda to establish our knowledge base regarding meals and to delineate decision-making processes and rules congruent with our organizational goals.

3.1.2 Our Decision Model

As illustrated in Figure 3.1, our model, designed to reduce the number of meals displayed to the user, constitutes the core of our system. It begins with the user inputting their preferences: whether they are carnivorous or vegetarian, their level of calorie-consciousness, any specific allergies (such as lactose, gluten, or both), and the course (appetizer, first course, etc.). Additionally, there are supplementary pieces of information that the user can request, which do not affect the core model but are nonetheless provided to the user. These supplementary details include: the ingredients in a meal, the calories of these ingredients, and the total calories of the meal. In the following section, we will provide a detailed explanation of each table.

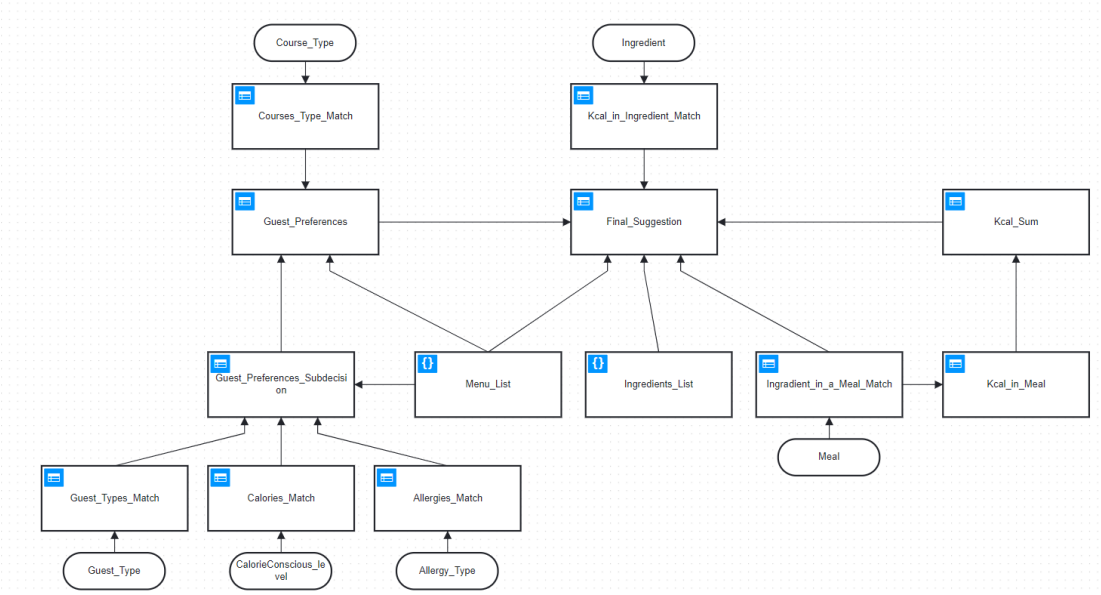

Figure 3.1: Our Decision Model in Camunda

3.1.3 Decision Tables

Now, referring to Figure 3.1, let's describe all the decision tables [3] and their respective inputs and outputs:

- *Guest_Types_Match*: It takes a string as input ("**Carnivorous**"; "**Vegetarian**"; "") and returns a list of meals filtered based on the user's choice. Note that if the user inputs an empty string, as mentioned earlier, the complete list of meals will be returned. This ensures that the user's options are not unnecessarily limited during interactions with the system.

[2] Camunda Documentation https://docs.camunda.org/manual/7.19/reference/dmn/feel/

[3] Decision Table: https://en.wikipedia.org/wiki/Decision_table

- *Calories_Match*: It takes an **integer** input, defined between 0 and 3 inclusive, and returns a list of meals filtered according to the number entered by the user. The default value is set to 0. A level 0 indicates that the user is not concerned about the calorie content of the meal. Level 1 means the user is slightly concerned about the calories in each meal, level 2 means the user is quite concerned, and level 3 indicates that the user is very concerned about the calorie content. Therefore, at level 3, a list of low-calorie meals will be returned.

- *Allergies_Match*: It takes a string as input ("**Lactose**"; "**Gluten**"; "**Lactose and Gluten**"; "") and returns a list of meals filtered based on the user's choice. If the user has a lactose intolerance, meals containing lactose, such as cheese, will not be displayed.

- *Guest_Preferences_Subdecision*: It takes as input the results produced by the three tables described above and returns a list based on the user's preferences. Essentially, it performs a logical AND operation, returning a list that includes only those meals present in all three input lists.

- *Courses_Type_Match*: It takes as input a type of course (including drinks) and returns a list of meals that include that course. If the user does not specify any preference, the entire menu is considered.

- *Guest_Preferences*: Essentially, it takes the previously filtered menu list and compares it with the list of courses selected by the user. This approach was chosen primarily for the sake of table readability; otherwise, we would have ended up with a 16-row table where the filtering mechanism would not have been immediately clear.

- *Ingredient_in_a_Meal_Match*: It takes a string as input (e.g., "Lasagne"; "Rigatoni alla Carbonara"; ...; "") and returns a list of ingredients contained in the specified meal. If the user has not entered anything (i.e., the input string is empty), an empty list will be returned.

- *Kcal_in_Meal*: It is based on the user's previous choice. If the user has selected a meal for which they want information, the calorie content of that meal will also be calculated. Essentially, it takes the list passed from the previous table as input, and if the ingredient is present, the calories will be recorded step by step.

- *Kcal_Sum*: It calculates the total calories for the meal.

- *Kcal_in_Ingredient_Match*: It takes a string as input (e.g., "Oil"; "Pasta"; ...; "") and returns an integer representing the number of calories in an average serving of that ingredient per meal.

- *Final_Suggestion*: This is the final decision table, which essentially suggests to the user which type of meal(s) may be most suitable based on their preferences and the menu's availability. Additionally, it can provide information about the ingredients contained in a specific meal and the corresponding calorie content (both of the ingredients and the meals).

3.1.4 Literal Expression

To simplify our process, we have developed a literal expression that encapsulates the Menu and the Ingredient list in the form of a string. These are represented in Figures 3.2 and 3.3, respectively.

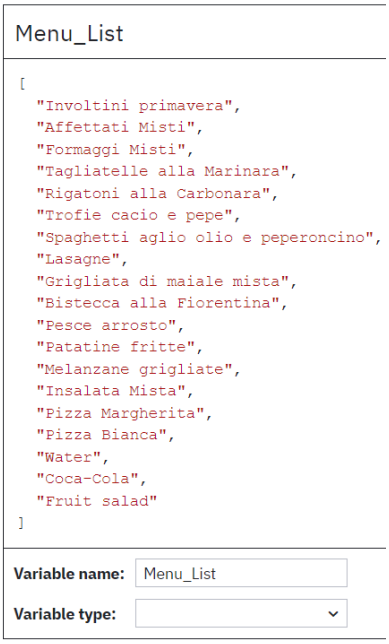

Figure 3.2: Menu List Literal Expression

Figure 3.3: Ingredient List Literal Expression

3.1.5 Camunda Simulation Example

Finally, we present an example of meal suggestions from our model using the Camunda DMN Simulator [4].

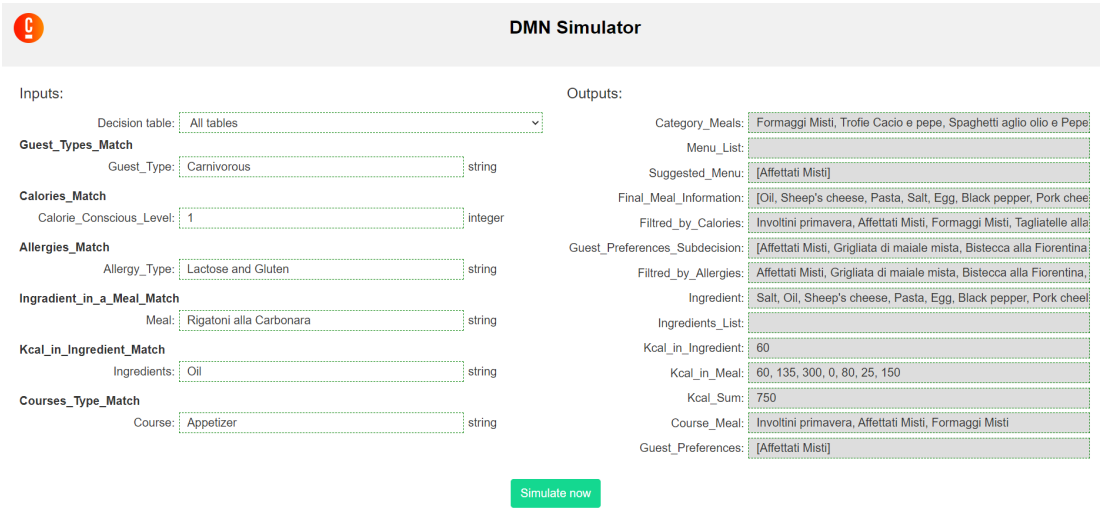

Figure 3.4: DMN Simulation

[4]DMN Simulator Reference Page: https://consulting.camunda.com/dmn-simulator/

3.2 Prolog

Logic programming utilizes logic as a declarative representation language and backward chaining as an inference rule. Prolog[5] is a logic programming language associated with artificial intelligence and computational linguistics. The strength of this language lies in its logical foundation. Prolog syntax provides:

- **Symbols**: **,** (and) **;** (or) **:-** (if) **not** (not);
- **Variables**: a variable in Prolog is a string of letters, digits, and underscores (_) beginning either with a capital letter or with an underscore;
- **Facts**: a fact is a predicate expression that makes a declarative statement about the problem domain;
- **Rules**: a rule is a predicate expression that uses logical implication (**:-**) to describe a relationship among facts. Thus a Prolog rule takes the form:

$$left\_hand\_side \text{ :- } right\_hand\_side \text{ .}$$

- **Queries**: the Prolog interpreter answers queries on the facts and rules represented in its database. By asking a question, Prolog is asked whether it can prove that the question is true. If the answer is "yes", Prolog answers "yes" and displays all the links between the variables made to obtain the answer. If it cannot prove that the query is true, it answers "no".;

Furthermore, in Prolog, the technique of backtracking is employed. Backtracking is a process where Prolog examines the truthfulness of various predicates by evaluating their correctness. Essentially, the Prolog interpreter explores different potential paths through backtracking until it discovers a valid route that reaches the final destination. Additionally, there are several built-in predicates, i.e., native rules provided by Prolog, such as "length(?List, ?Length)" which allows obtaining the length of a list. Prolog is so powerful that it allows defining a knowledge base in just a few lines. We leveraged this capability to create our rule-based system for meals and ingredients.

3.2.1 Prolog Implementation

We have implemented the model defined in the DMN file using Prolog and tested our implementation on the site [6]. Prolog code is described below.

```
% Facts related to guest types and associated meals
meal(carnivorous, formaggi_misti).
meal(vegetarian, formaggi_misti).
meal(carnivorous, trofie_cacio_e_pepe).
meal(vegetarian, trofie_cacio_e_pepe).
meal(carnivorous, spaghetti_aglio_olio_e_peperoncino).
meal(vegetarian, spaghetti_aglio_olio_e_peperoncino).
meal(carnivorous, patatine_fritte).
meal(vegetarian, patatine_fritte).
meal(carnivorous, pizza_margherita).
meal(vegetarian, pizza_margherita).
meal(carnivorous, pizza_bianca).
meal(vegetarian, pizza_bianca).
```

[5] Prolog Wikipedia: https://it.wikipedia.org/wiki/Prolog
[6] Prolog Testing: https://swish.swi-prolog.org/

```
meal(carnivorous, water).
meal(vegetarian, water).
meal(carnivorous, coca-cola).
meal(vegetarian, coca-cola).
meal(carnivorous, affettati_misti).
meal(carnivorous, tagliatelle_alla_marinara).
meal(carnivorous, lasagne).
meal(carnivorous, grigliata_di_maiale_mista).
meal(carnivorous, bistecca_alla_fiorentina).
meal(carnivorous, pesce_arrosto).
meal(vegetarian, involtini_primavera).
meal(vegetarian, melanzane_grigliate).
meal(vegetarian, insalata_mista).
meal(vegetarian, fruit_salad).
```

Listing 3.1: Facts for Meals

This initial Prolog code defines a series of facts related to different types of guests and the meals associated with them. Specifically, it categorizes various meals as suitable for either carnivorous or vegetarian guests. Each fact follows the structure meal(Type, Meal), where Type indicates whether the meal is appropriate for a carnivorous or vegetarian guest, and Meal is the name of the dish.

```
% Rule to get the meal based on the guest type
category_meals(GuestType, Meal) :- meal(GuestType, Meal).
```

Listing 3.2: Rule for Meals

This Prolog rule 3.2.1 defines a method to retrieve meals based on the type of guest.

```
% Facts related to calories and associated meals
filtered_by_calories(Level, involtini_primavera) :- between(0, 3, Level).
filtered_by_calories(Level, affettati_misti) :- between(0, 2, Level).
filtered_by_calories(Level, formaggi_misti) :- between(0, 1, Level).
filtered_by_calories(Level, tagliatelle_alla_marinara) :- between(0, 2,
    Level).
filtered_by_calories(Level, rigatoni_alla_carbonara) :- between(0, 1, Level)
    .
filtered_by_calories(Level, trofie_cacio_e_pepe) :- between(0, 2, Level).
filtered_by_calories(Level, spaghetti_aglio_olio_e_peperoncino) :- between
    (0, 3, Level).
filtered_by_calories(Level, lasagne) :- between(0, 1, Level).
filtered_by_calories(Level, grigliata_di_maiale_mista) :- between(0, 1,
    Level).
filtered_by_calories(Level, bistecca_alla_fiorentina) :- between(0, 2, Level
    ).
filtered_by_calories(Level, pesce_arrosto) :- between(0, 3, Level).
filtered_by_calories(Level, patatine_fritte) :- Level = 0.
filtered_by_calories(Level, melanzane_grigliate) :- between(0, 3, Level).
filtered_by_calories(Level, insalata_mista) :- between(0, 3, Level).
filtered_by_calories(Level, pizza_margherita) :- between(0, 1, Level).
filtered_by_calories(Level, pizza_bianca) :- between(0, 2, Level).
filtered_by_calories(Level, water) :- between(0, 3, Level).
filtered_by_calories(Level, coca-cola) :- between(0, 1, Level).
filtered_by_calories(Level, fruit_salad) :- between(0, 2, Level).
```

Listing 3.3: Facts for Filtered by Calories

This Prolog code 3.2.1 defines facts related to the caloric content of various meals, categorizing them based on different calorie levels. Each fact follows the structure filtered_by_calories(Level, Meal), where Level represents a range of caloric levels, and

Meal is the name of the dish. The between/3 predicate is used to specify the range of calorie levels for each meal.

```
% Rule to get meals filtered based on your level of calorie awareness
calories_match(Level, Meal) :- filtered_by_calories(Level, Meal).
```

Listing 3.4: Rule for Filtered by Calories

This Prolog rule 3.2.1 defines a method for retrieving meals based on a specified level of calorie awareness.

```
% Meal facts filtered by allergy type
filtered_by_allergies(lactose, affettati_misti).
filtered_by_allergies(gluten, affettati_misti).
filtered_by_allergies(lactose_and_gluten, affettati_misti).
filtered_by_allergies(gluten, formaggi_misti).
filtered_by_allergies(lactose, tagliatelle_alla_marinara).
filtered_by_allergies(lactose, spaghetti_aglio_olio_e_peperoncino).
filtered_by_allergies(lactose, grigliata_di_maiale_mista).
filtered_by_allergies(gluten, grigliata_di_maiale_mista).
filtered_by_allergies(lactose_and_gluten, grigliata_di_maiale_mista).
filtered_by_allergies(lactose, bistecca_alla_fiorentina).
filtered_by_allergies(gluten, bistecca_alla_fiorentina).
filtered_by_allergies(lactose_and_gluten, bistecca_alla_fiorentina).
filtered_by_allergies(lactose, pesce_arrosto).
filtered_by_allergies(gluten, pesce_arrosto).
filtered_by_allergies(lactose_and_gluten, pesce_arrosto).
filtered_by_allergies(lactose, patatine_fritte).
filtered_by_allergies(gluten, patatine_fritte).
filtered_by_allergies(lactose_and_gluten, patatine_fritte).
filtered_by_allergies(lactose, melanzane_grigliate).
filtered_by_allergies(gluten, melanzane_grigliate).
filtered_by_allergies(lactose_and_gluten, melanzane_grigliate).
filtered_by_allergies(lactose, insalata_mista).
filtered_by_allergies(gluten, insalata_mista).
filtered_by_allergies(lactose_and_gluten, insalata_mista).
filtered_by_allergies(lactose, pizza_bianca).
filtered_by_allergies(lactose, water).
filtered_by_allergies(gluten, water).
filtered_by_allergies(lactose_and_gluten, water).
filtered_by_allergies(lactose, coca-cola).
filtered_by_allergies(gluten, coca-cola).
filtered_by_allergies(lactose_and_gluten, coca-cola).
filtered_by_allergies(lactose, fruit_salad).
filtered_by_allergies(gluten, fruit_salad).
filtered_by_allergies(lactose_and_gluten, fruit_salad).
```

Listing 3.5: Facts for Filtered by Allergies

This Prolog code 3.2.1defines a series of facts related to meal options filtered by allergy type. Each fact follows the structure *filtered_by_allergies(AllergyType, Meal)*, where *AllergyType* indicates the type of allergy (**lactose**, **gluten**, or **both**), and *Meal* is the name of the dish.

```
% Rule to get filtered meals based on allergy type
allergies_match(Allergy_Type, Meal) :- filtered_by_allergies(Allergy_Type,
    Meal).
```

Listing 3.6: Rule for Filtered by Allergies

This Prolog rule defines a method for retrieving meals based on a specified allergy type.

```
% Meal facts releted to courses
```

```
filtered_by_courses(appetizer, involtini_primavera).
filtered_by_courses(appetizer, affettati_misti).
filtered_by_courses(appetizer, formaggi_misti).
filtered_by_courses(first_meal, spaghetti_aglio_olio_e_peperoncino).
filtered_by_courses(first_meal, tagliatelle_alla_marinara).
filtered_by_courses(first_meal, lasagne).
filtered_by_courses(first_meal, rigatoni_alla_carbonara).
filtered_by_courses(first_meal, trofie_cacio_e_pepe).
filtered_by_courses(second_meal, grigliata_di_maiale_mista).
filtered_by_courses(second_meal, bistecca_alla_fiorentina).
filtered_by_courses(second_meal, pesce_arrosto).
filtered_by_courses(sidedish, patatine_fritte).
filtered_by_courses(sidedish, insalata_mista).
filtered_by_courses(sidedish, melanzane_grigliate).
filtered_by_courses(pizza, pizza_margherita).
filtered_by_courses(pizza, pizza_bianca).
filtered_by_courses(dessert, fruit_salad).
filtered_by_courses(drink, water).
filtered_by_courses(drink, coca_cola).
```

Listing 3.7: Facts for Filtered by Courses

This Prolog code defines a series of facts related to the categorization of meals based on their courses. Each fact follows the structure *filtered_by_courses(CourseType, Meal)*, where *CourseType* represents the type of course (e.g., appetizer, first meal, second meal, sidedish, pizza, dessert, drink), and *Meal* is the name of the dish.

```
% Rule to get filtered meals based on courses
courses_match(Course_Type, Meal) :- filtered_by_courses(Course_Type, Meal).
```

Listing 3.8: Rule for Filtered by Courses

This Prolog rule defines a method for retrieving meals based on a specified course type.

```
% Rule to determine guest preferences based on category, calories, allergies
    , and courses
guest_preferences(none, CalorieLevel, none, none, GuestPreferences) :-
    once(findall(Meal, (calories_match(CalorieLevel, Meal)),
    GuestPreferences)).

guest_preferences(none, CalorieLevel, none, Course_Type, GuestPreferences)
    :-
    once(findall(Meal, (calories_match(CalorieLevel, Meal), courses_match(
    Course_Type, Meal)), GuestPreferences)).

guest_preferences(none, CalorieLevel, AllergyType, none, GuestPreferences)
    :-
    once(findall(Meal, (calories_match(CalorieLevel, Meal), allergies_match(
    AllergyType, Meal)), GuestPreferences)).

guest_preferences(none, CalorieLevel, AllergyType, Course_Type,
    GuestPreferences) :-
    once(findall(Meal, (calories_match(CalorieLevel, Meal), allergies_match(
    AllergyType, Meal), courses_match(Course_Type, Meal)), GuestPreferences)
    ).

guest_preferences(GuestType, CalorieLevel, none, none, GuestPreferences) :-
    once(findall(Meal, (category_meals(GuestType, Meal), calories_match(
    CalorieLevel, Meal)), GuestPreferences)).

guest_preferences(GuestType, CalorieLevel, none, Course_Type,
    GuestPreferences) :-
```

```
18      once(findall(Meal, (category_meals(GuestType, Meal), calories_match(
        CalorieLevel, Meal), courses_match(Course_Type, Meal)), GuestPreferences
        )).
19
20  guest_preferences(GuestType, CalorieLevel, Allergy_Type, none,
        GuestPreferences) :-
21      once(findall(Meal, (category_meals(GuestType, Meal), calories_match(
        CalorieLevel, Meal), allergies_match(Allergy_Type, Meal)),
        GuestPreferences)).
22
23  guest_preferences(GuestType, CalorieLevel, AllergyType, Course_Type,
        GuestPreferences) :-
24      once(findall(Meal, (category_meals(GuestType, Meal), calories_match(
        CalorieLevel, Meal), allergies_match(AllergyType, Meal), courses_match(
        Course_Type, Meal)), GuestPreferences)).
```

Listing 3.9: Rules for Guest Preferences

This Prolog code represent the core of the program and defines a set of rules to determine guest preferences based on various criteria such as guest type, calorie level, allergy type, and meal courses. Each *guest_preferences* rule takes inputs for guest type, calorie level, allergy type, meal course type, and returns a list of meals that match the specified preferences.

```
1  % Facts related to ingredients for each dish
2  ingredient(involtini_primavera, salt).
3  ingredient(affettati_misti, salt).
4  ingredient(formaggi_misti, salt).
5  ingredient(tagliatelle_alla_marinara, salt).
6  ingredient(rigatoni_alla_carbonara, salt).
7  ingredient(trofie_cacio_e_pepe, salt).
8  ingredient(spaghetti_aglio_olio_e_peperoncino, salt).
9  ingredient(lasagne, salt).
10 ingredient(grigliata_di_maiale_mista, salt).
11 ingredient(bistecca_alla_fiorentina, salt).
12 ingredient(pesce_arrosto, salt).
13 ingredient(patatine_fritte, salt).
14 ingredient(melanzane_grigliate, salt).
15 ingredient(insalata_mista, salt).
16 ingredient(pizza_margherita, salt).
17 ingredient(pizza_bianca, salt).
18
19 ingredient(involtini_primavera, oil).
20 ingredient(tagliatelle_alla_marinara, oil).
21 ingredient(rigatoni_alla_carbonara, oil).
22 ingredient(trofie_cacio_e_pepe, oil).
23 ingredient(spaghetti_aglio_olio_e_peperoncino, oil).
24 ingredient(lasagne, oil).
25 ingredient(bistecca_alla_fiorentina, oil).
26 ingredient(pesce_arrosto, oil).
27 ingredient(patatine_fritte, oil).
28 ingredient(melanzane_grigliate, oil).
29 ingredient(insalata_mista, oil).
30 ingredient(pizza_margherita, oil).
31 ingredient(pizza_bianca, oil).
32
33 ingredient(involtini_primavera, carrot).
34 ingredient(lasagne, carrot).
35
36 ingredient(involtini_primavera, cabbage).
37
```

```
ingredient(affettati_misti, ciauscolo).
ingredient(affettati_misti, salame).
ingredient(affettati_misti, ham).

ingredient(formaggi_misti, cow_cheese).
ingredient(pizza_margherita, cow_cheese).

ingredient(formaggi_misti, sheep_cheese).
ingredient(rigatoni_alla_carbonara, sheep_cheese).

ingredient(tagliatelle_alla_marinara, pasta).
ingredient(rigatoni_alla_carbonara, pasta).
ingredient(trofie_cacio_e_pepe, pasta).
ingredient(spaghetti_aglio_olio_e_peperoncino, pasta).
ingredient(lasagne, pasta).

ingredient(tagliatelle_alla_marinara, shrimp).
ingredient(pesce_arrosto, shrimp).

ingredient(involtini_primavera, puff_pastry).

ingredient(tagliatelle_alla_marinara, mussel).
ingredient(pesce_arrosto, mussel).

ingredient(formaggi_misti, parmesan).
ingredient(lasagne, parmesan).

ingredient(rigatoni_alla_carbonara, egg).

ingredient(affettati_misti, black_pepper).
ingredient(rigatoni_alla_carbonara, black_pepper).
ingredient(trofie_cacio_e_pepe, black_pepper).

ingredient(rigatoni_alla_carbonara, pork_cheek).

ingredient(spaghetti_aglio_olio_e_peperoncino, garlic).

ingredient(spaghetti_aglio_olio_e_peperoncino, chili).

ingredient(lasagne, tomato).
ingredient(pizza_margherita, tomato).

ingredient(lasagne, bechamel).

ingredient(lasagne, minced_meat).

ingredient(grigliata_di_maiale_mista, pork_rib).
ingredient(grigliata_di_maiale_mista, sausage).
ingredient(grigliata_di_maiale_mista, pork_chop).

ingredient(bistecca_alla_fiorentina, steak).

ingredient(pesce_arrosto, calamari).
ingredient(pesce_arrosto, sword_fish).
ingredient(pesce_arrosto, salmon).

ingredient(patatine_fritte, potato).

ingredient(melanzane_grigliate, eggplant).

ingredient(pizza_margherita, water).
```

```
ingredient(pizza_bianca, water).
ingredient(coca_cola, water).

ingredient(insalata_mista, lettuce).
ingredient(insalata_mista, corn).

ingredient(pizza_margherita, type_0_flour).
ingredient(pizza_bianca, type_0_flour).

ingredient(coca_cola, sugar_and_food_coloring).

ingredient(fruit_salad, apple).
ingredient(fruit_salad, banana).
ingredient(fruit_salad, strawberry).
```

Listing 3.10: Facts for Ingredients contained in a Meal

This Prolog dataset outlines the ingredients associated with various dishes. Each dish is meticulously categorized by its constituent elements, illustrating a diverse range of culinary components: The dishes encompass a spectrum of flavors and preparations. For example, pasta-based dishes such as "tagliatelle alla marinara" and "rigatoni alla carbonara" feature ingredients like pasta itself—varying in type—and seafood such as shrimp and mussels. These are complemented by sauces like bechamel and seasonings such as black pepper. Meat-focused options such as "grigliata di maiale mista" and "bistecca alla fiorentina" emphasize grilled meats, including pork ribs, sausages, pork chops, and steak, ensuring a robust carnivorous offering. Vegetarian dishes such as "melanzane grigliate" and "insalata mista" showcase grilled eggplant and fresh salad components like lettuce and corn, highlighting a commitment to both flavor and variety. Traditional Italian pizzas like "pizza margherita" and "pizza bianca" incorporate essentials like tomatoes, type 0 flour for the crust, and cheeses such as cow cheese and parmesan, maintaining a timeless culinary tradition. Beverages like "coca cola" and "water" are included, with "coca cola" containing sugar and food coloring as fundamental ingredients. For dessert, "fruit salad" is composed of a medley of fruits such as apples, bananas, and strawberries, providing a refreshing conclusion to the meal. This dataset not only underscores the richness of Italian cuisine but also demonstrates a meticulous attention to detail in capturing the essence of each dish through its specific ingredients.

```
% Rule to find the ingredients of a dish
dish_ingredients(Meal, Ingredients) :-
    findall(Ingredient, ingredient(Meal, Ingredient), Ingredients).
```

Listing 3.11: Rule for Ingredients contained in a Meal

This Prolog rule, *dish_ingredients*, is designed to retrieve the list of ingredients (**Ingredients**) associated with a given dish (**Meal**)

```
% Calories for single ingredient
kcal_in_ingredient(oil, 60).
kcal_in_ingredient(eggplant, 18).
kcal_in_ingredient(cows_cheese, 115).
kcal_in_ingredient(sheeps_cheese, 135).
kcal_in_ingredient(pasta, 300).
kcal_in_ingredient(shrimp, 35).
kcal_in_ingredient(ciauscolo, 70).
kcal_in_ingredient(salame, 50).
kcal_in_ingredient(ham, 45).
kcal_in_ingredient(carrot, 27).
```

```
kcal_in_ingredient(cabbage, 25).
kcal_in_ingredient(puff_pastry, 150).
kcal_in_ingredient(mussel, 40).
kcal_in_ingredient(salt, 0).
kcal_in_ingredient(water, 0).
kcal_in_ingredient(parmesan, 75).
kcal_in_ingredient(egg, 80).
kcal_in_ingredient(black_pepper, 25).
kcal_in_ingredient(pork_cheek, 150).
kcal_in_ingredient(garlic, 3).
kcal_in_ingredient(chili, 15).
kcal_in_ingredient(tomato, 30).
kcal_in_ingredient(bechamel, 105).
kcal_in_ingredient(mozzarella_cheese, 150).
kcal_in_ingredient(minced_meat, 120).
kcal_in_ingredient(pork_rib, 70).
kcal_in_ingredient(sausage, 205).
kcal_in_ingredient(pork_chop, 125).
kcal_in_ingredient(steak, 485).
kcal_in_ingredient(calamari, 50).
kcal_in_ingredient(sword_fish, 120).
kcal_in_ingredient(salmon, 150).
kcal_in_ingredient(potato, 450).
kcal_in_ingredient(lettuce, 25).
kcal_in_ingredient(mais, 50).
kcal_in_ingredient(sugar_and_food_coloring, 204).
kcal_in_ingredient(apple, 35).
kcal_in_ingredient(banana, 40).
kcal_in_ingredient(strawberry, 22).
kcal_in_ingredient(_, 0).  % Default value for any undefined ingredient
```

Listing 3.12: Facts for Kcalories in an Ingredient

The Prolog facts you provided associate specific ingredients with their respective calorie values using the kcal_in_ingredient predicate.

```
% Rule to obtain the calories of a single ingredient
ingredient_kcal(Ingredient, Kcal) :-
    once(kcal_in_ingredient(Ingredient, Kcal)).

% Calculation of the total calories of a meal given a list of ingredients
kcal_in_meal([], 0).
kcal_in_meal([Ingredient | Rest], TotalKcal) :-
    kcal_in_ingredient(Ingredient, Kcal),
    kcal_in_meal(Rest, RestKcal),
    TotalKcal is Kcal + RestKcal.

% Calculation of the total calories of a meal given the name of the meal
meal_total_kcal(Meal, TotalKcal) :-
    once(dish_ingredients(Meal, Ingredients)),
    once(kcal_in_meal(Ingredients, TotalKcal)).

% Rule to obtain both the ingredients and the total calories of a dish
meal_info(Meal, Ingredients, TotalKcal) :-
    once(dish_ingredients(Meal, Ingredients)),
    once(meal_total_kcal(Meal, TotalKcal)).
```

Listing 3.13: Rules for Meal Info

The following rules are explained below:

1. The first rule retrieves the calorie content (Kcal) of a specified Ingredient using the kcal_in_ingredient predicate.

2. The second rule computes the total calorie count (TotalKcal) of a meal represented as a list of Ingredients. It recursively sums up the calories of each Ingredient in the list using kcal_in_ingredient.

3. The third rule calculates the total calorie count (TotalKcal) of a meal specified by its Meal name. It first retrieves the list of Ingredients for the meal using dish_ingredients and then calculates the total calories using kcal_in_meal.

4. Finally, the fourth rule combines the retrieval of both Ingredients and TotalKcal for a given Meal. It ensures that both pieces of information are obtained once and only once, using once to avoid unnecessary backtracking.

Prolog Testing

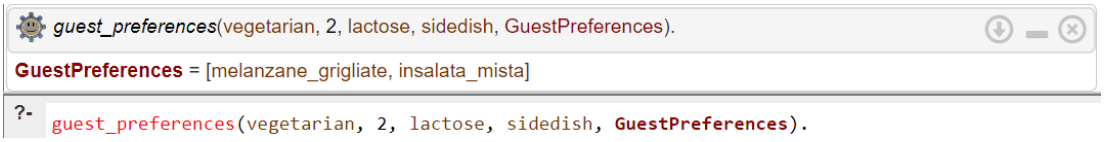

Figure 3.5: Prolog Testing 1

In this scenario, if the user is vegetarian, has a moderate to high calorie awareness, is lactose intolerant, and desires a side dish, the system returns the following dishes: 'melanzane_grigliate' and 'insalata_mista'.

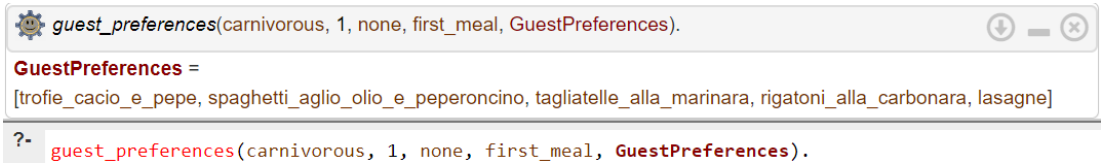

Figure 3.6: Prolog Testing 2

In this case, if I am carnivorous, have a moderate to low calorie awareness per meal, no allergies, and I am interested in a first course, the following meals are returned: 'trofie_cacio_e_pepe', 'spaghetti_aglio_olio_e_peperoncino', 'tagliatelle_alla_marinara', 'rigatoni_alla_carbonara', 'lasagne'.

Here, we provide example queries to run in the simulator described before for other results.

```
% guest_preferences(carnivorous, 2, lactose, none, GuestPreferences).

% guest_preferences(none, 0, none, none, GuestPreferences).

% guest_preferences(none, 0, lactose, appetizer, GuestPreferences).

% ingredient_kcal(carrot, Kcal).

% meal_info(lasagne, Ingredients, TotalKcal).
```

Listing 3.14: Queries for Prolog Testing

3.3 Ontology Engineering

In Computer Science, ontology engineering is a discipline focused on methodologies for constructing ontologies. This process involves formally representing and defining categories, properties, and relationships among concepts, data, and entities. Ontologies can be structured using various data models such as RDF, OWL, Neo4J, and others. For our ontology construction, we employed RDF.

RDF

The Resource Description Framework (**RDF**) is a standardized approach utilized for describing and exchanging graph data in a general manner. RDF enables the description of resources that are utilized in constructing knowledge graphs.
A **Knowledge Graph** depicts a network of real-world entities such as objects, events, situations, or concepts, highlighting the relationships that exist between them.

Protégé

Protégé is a freely available, open-source ontology editor and knowledge management system. It enables the definition of Semantic Web Rule Language (**SWRL**) to establish inference rules that generate new data from existing ontologies. Additionally, Protégé supports the Sparql Protocol And Rdf Query Language (**SPARQL**), a semantic query language designed for databases capable of retrieving and manipulating data stored in RDF format.

3.3.1 Our Ontology

Utilizing Protégé, we have developed our ontology by incorporating the primary entities.

Classes

The primary entities defined earlier also serve as our main classes. Additionally, we have established a hierarchy to facilitate the future inclusion of other resources, such as potential allergens, types of ingredients, courses, and more.

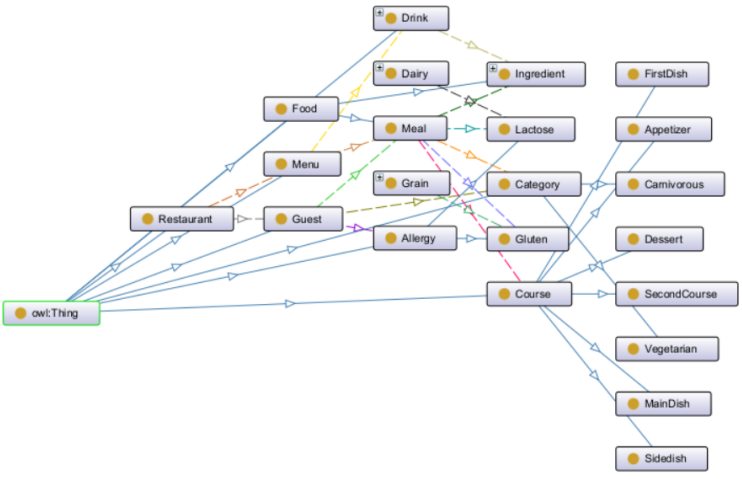

Figure 3.7: Our Ontology Graph in Protégé

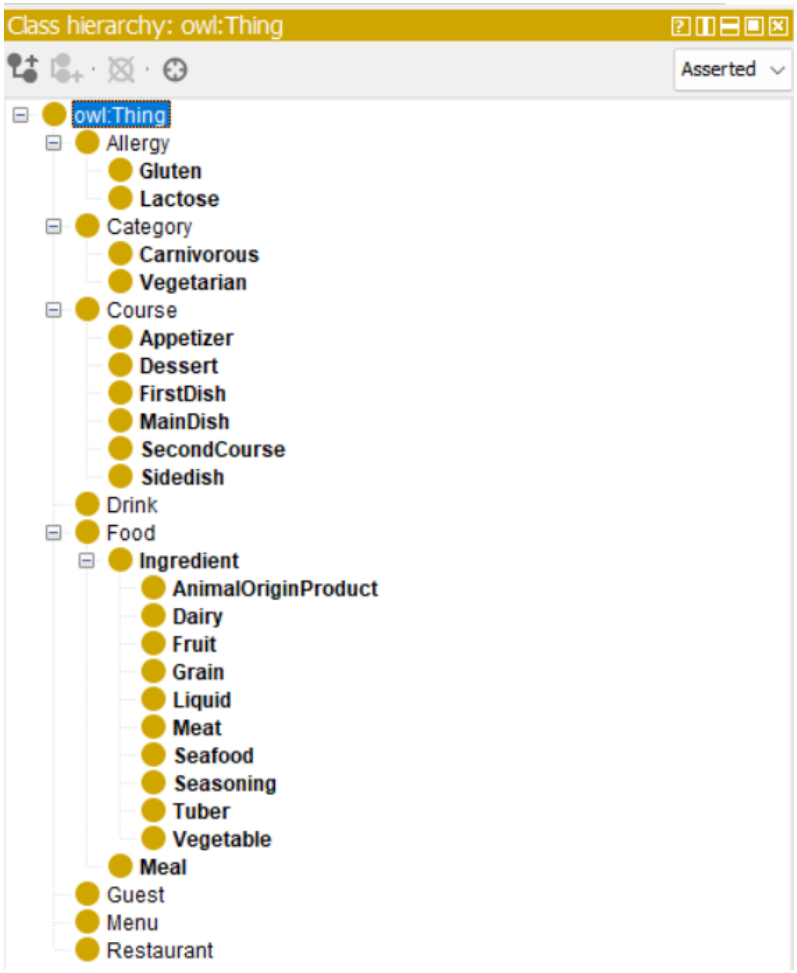

Figure 3.8: Our Ontology Model in Protégé

Object Properties

Object properties are used to represent the relationships between classes. We defined several object properties for the classes:

- **restaurant_hasMenu**: to associate a menu with a restaurant;
- **restaurant_hasGuest**: to associate guest with a restaurant;
- **menu_containsDrink**:to associate a drink with a menu;
- **menu_containsMeal**: to associate a meal with a menu;
- **guest_hasAllergies**: it represents a guest that has an allergy;
- **guest_hasCategory**: to associate a category (e.g. Carnivorous) with a guest;
- **guest_hasPreferenceCourse**: to associate a specific course with a guest;
- **guest_isAtRiskForFood**: this is a SWRL rule inferred by the reasoner, which is useful for identifying potential meal-related risks due to allergies;

- **meal_containsGlutenIntolerance**: this is an SWRL rule inferred by the reasoner, which identifies if the meal contains ingredients with gluten (e.g., grain);
- **meal_containsLactoseIntolerance**: this is an SWRL rule inferred by the reasoner, which identifies if the meal contains ingredients with lactose (e.g., dairy);
- **meal_hasCategory**: to associate a category with the meal;
- **meal_hasCourse**: to associate a course with the meal;
- **meal_hasIngredient**: to associate an ingredient with a meal;
- **drink_hasIngredient**: to associate an ingredient with a drink;
- **dairy_containsLactoseIntolerance**; to associate the lactose intolerance to class dairy;
- **grain_containsGlutenIntolerance**: to associate the gluten intolerance to class grain.

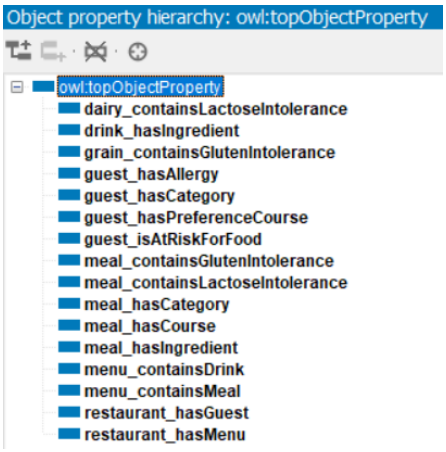

Figure 3.9: Object Properties for our Model

Data Properties

Data Properties represent the attributes of the classes. Below, all the Data Properties are illustrated:

- **thing_hasName**: this is an attribute that all subclasses of the class Thing possess. It is a string used to identify the name of the object;
- **food_hasKcal**: this is an attribute that all subclasses of the class Food (in this case we have only Ingredient and Meal) posses. It is an integer used to identify the kcal of the food;
- **guest_hasLevelOfCalorieConscious**: this is an attribute for the class Guest that represents the level of calorie-consciousness of a guest. It is an integer ranging between 0 and 3;

- **meal_hasLevelOfCalorieConscious**: this is an attribute for the class Meal that represents the level of calorie-consciousness of a meal. It is an integer ranging between 0 and 3;
- **drink_hasKcal**: this is an attribute that the class Drink possesses. It is an integer used to identify the kcal of the drink;

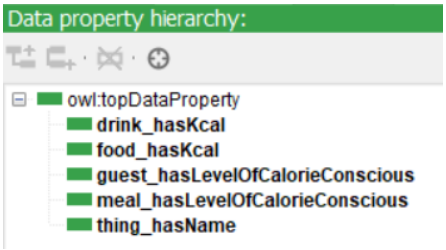

Figure 3.10: Data Properties for our Model

Individuals

To reduce space, we have omitted the inclusion of all individuals present in the classes in our knowledge base in this report, even though they have been defined.

3.3.2 SWRL Rules

We have exploited the reasoner provided by Protégé its power by defining the following rules:

GuestIsAtRiskForFoodGluten

$$\begin{aligned}&\text{Guest}(?guest) \wedge \text{guest\_hasAllergy}(?guest, ?allergy)\\ &\quad \wedge \text{Meal}(?meal) \wedge \text{meal\_containsGlutenIntolerance}(?meal, ?allergy)\\ &\quad \rightarrow \text{guest\_isAtRiskForFood}(?guest, ?meal)\end{aligned}$$

This rule is designed to infer meals that pose a risk to the user based on whether the guest has an allergy, in this case gluten intolerance, or not, and to display them accordingly.

GuestIsAtRiskForFoodLactose

$$\begin{aligned}&\text{Guest}(?guest) \wedge \text{guest\_hasAllergy}(?guest, ?allergy)\\ &\quad \wedge \text{Meal}(?meal) \wedge \text{meal\_containsLactoseIntolerance}(?meal, ?allergy)\\ &\quad \rightarrow \text{guest\_isAtRiskForFood}(?guest, ?meal)\end{aligned}$$

This rule is designed to infer meals that pose a risk to the user based on whether the guest has an allergy, in this case lactose intolerance, or not, and to display them accordingly.

MealContainsGlutenIntolerance

$$\begin{aligned}&\text{Meal}(?meal) \land \text{meal\_hasIngredient}(?meal, ?ingredient)\\ &\quad \land \text{Ingredient}(?ingredient) \land \text{grain\_containsGlutenIntolerance}(?ingredient, ?gluten)\\ &\rightarrow \text{meal\_containsGlutenIntolerance}(?meal, ?gluten)\end{aligned}$$

This rule is used to infer meals that contain ingredients which include a gluten intolerance. Therefore, the reasoning is: if the meal contains an ingredient that has a gluten intolerance, then the meal itself has a gluten intolerance.

MealContainsLactoseIntolerance

$$\begin{aligned}&\text{Meal}(?meal) \land \text{meal\_hasIngredient}(?meal, ?ingredient)\\ &\quad \land \text{Ingredient}(?ingredient) \land \text{dairy\_containsLactoseIntolerance}(?ingredient, ?gluten)\\ &\rightarrow \text{meal\_containsLactoseIntolerance}(?meal, ?gluten)\end{aligned}$$

This rule is used to infer meals that contain ingredients which include a lactose intolerance. Therefore, the reasoning is: if the meal contains an ingredient that has a lactose intolerance, then the meal itself has a lactose intolerance.

Unfortunately, it was not possible to further infer the data property field guest_hasLevelOfCalorieConscious and the field meal_hasLevelOfCalorieConscious with Reasoner HermiT 1.4.3.456 in a short time because it does not support SWRL built-in atoms. We did attempt to use the Pellet (incremental) Reasoner; however, sometimes it worked correctly and other times it resulted in errors. Therefore, we left the two fields without inferences, opting for a more rudimentary but still functional approach.

SWRL Testing

Below are examples of SWRL rule testing executed using Reasoner HermiT 1.4.3.456:

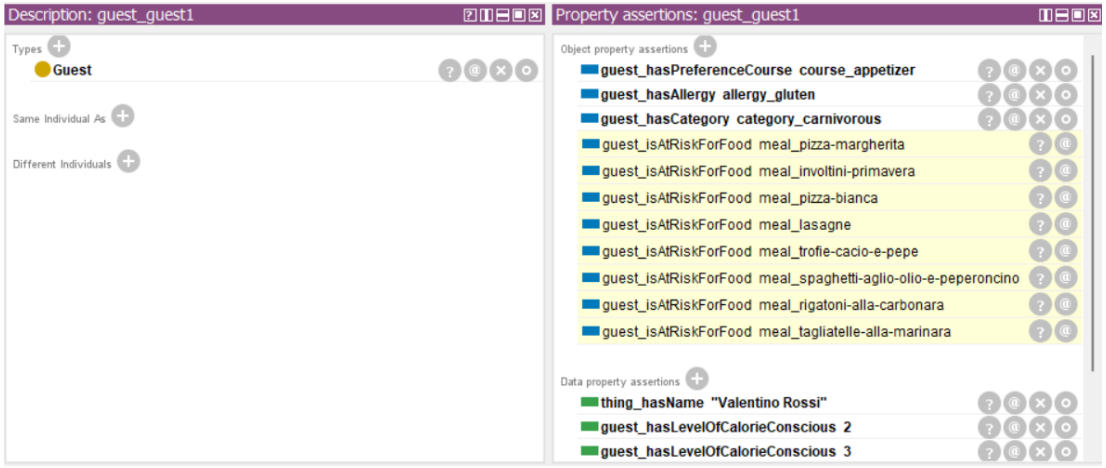

Figure 3.11: SWRL Testing 1 - Meal: Rigatoni all Carbonara

As depicted in the first figure (Figure 3.11), the high-risk dishes for the user are immediately inferred.

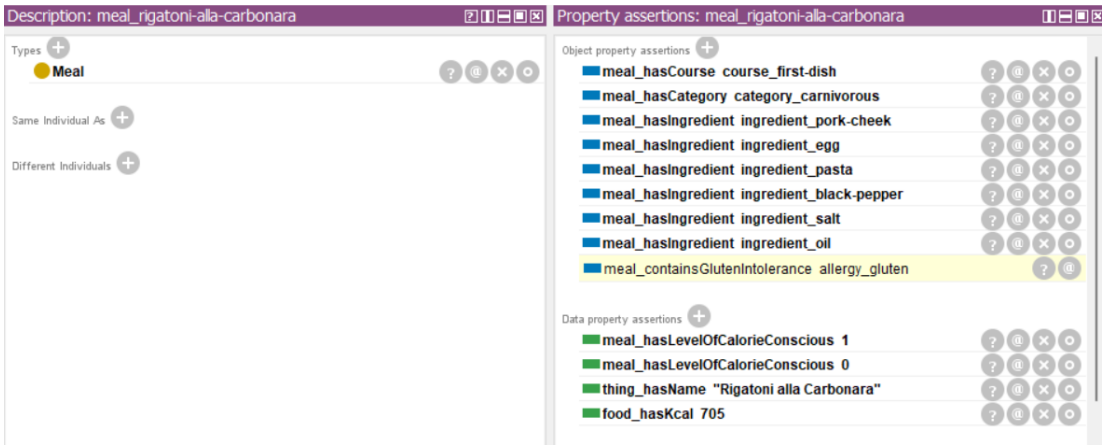

Figure 3.12: SWRL Testing 2 - Meal: Rigatoni all Carbonara

In this case, the inference is based on whether the meal contains an "allergenic" ingredient or not.

3.3.3 SPARQL Query

To save space, we do not describe the queries here, as they have already been documented in the file queries_sparql.txt[7].

3.3.4 SHACL Shape

The SHACL Shape File can be found in the 'ontology' folder and is named 'shacl_menu'[8]. Here is a description of the file:

- **RestaurantShape**: Ensures that instances of Restaurant have at least one and at most five menus (MenuShape). Optionally, it allows associations with guests (GuestShape).
- **MenuShape**: Defines constraints for instances of Menu, ensuring they contain between 1 and 300 meals (MealShape) and between 1 and 100 drinks (DrinkShape).
- **GuestShape**: Validates instances of Guest, specifying optional information such as allergies (AllergyShape), dietary categories (CategoryShape), food-related risks (MealShape), and a level of calorie consciousness.
- **MealShape**: Specifies constraints for instances of Meal, including optional intolerance to lactose (LactoseShape) and gluten (GlutenShape), categories, courses, and ingredients.
- **DrinkShape**: Defines constraints for instances of Drink, ensuring they have ingredients and specify caloric content.
- **AllergyShape** and **LactoseShape/GlutenShape**: These shapes define constraints for instances related to allergies and specific intolerances.
- **CategoryShape**: Ensures consistency in categorization within the ontology.

[7]SPARQL example queries: https://github.com/FrancescoChiocchi8/KEBI-Project/blob/main/Task1/ontology/queries_sparql.txt

[8]SHACL Shape File: https://github.com/FrancescoChiocchi8/KEBI-Project/blob/main/Task1/ontology/shacl_menu

4. Agile and Ontology-based Meta-Modelling

In Section 4.1, we will describe what AOAME is, while in Section 4.2, we will outline what BPMN 2.0 entails and present our model.

Here you can find the link to our project for Task 2 [1].

4.1 AOAME

AOAME [2] is a prototypical tool designed to implement an agile, ontology-based approach to meta-modeling. This approach aims to design and maintain schemas for Enterprise Knowledge Graphs (EKGs), which structure the knowledge of a specific application domain, enabling analysis, reasoning, and integration of information extracted from various data sources. The primary challenge in developing and maintaining an EKG schema is the requirement for expertise in both ontology engineering and the application domain.
The proposed approach extends traditional meta-modeling with agile principles, allowing for domain-specific adaptations of modeling languages and real-time testing. This integration facilitates the involvement of domain experts in the engineering cycle. AOAME, developed following the Design Science Research methodology, serves as a prototype to evaluate the utility of this approach. It provides capabilities to extend, modify, and remove modeling constructs, and to hide them from the tool palette. These actions are supported by meta-modeling operators that generate SPARQL statements and update the triplestore, ensuring consistency between human-interpretable and machine-interpretable knowledge.
The software architecture of AOAME leverages Java libraries of Fuseki to develop APIs that retrieve ontologies from the triplestore and display them in a GUI, as well as to implement the meta-modeling operators. This tool has been validated through the implementation of real-world scenarios and significant case studies, demonstrating its operability and generality across various application domains. AOAME facilitates close cooperation between domain experts and language engineers during the engineering of domain-specific modeling languages (DSMLs), thus enhancing the adaptability and effectiveness of the meta-modeling process.

[1] GitHub Forked Project link:
https://github.com/FrancescoChiocchi8/Ontology4ModelingEnvironment
[2] AOAME Reference: https://emisa-journal.org/emisa/article/view/310

4.2 BPMN 2.0

BPMN stands for Business Process Model and Notation. It is a graphical representation used to specify business processes in a workflow. BPMN provides a standardized way to visualize the sequence of business activities and the flow of information between them. It is designed to be understandable by all business stakeholders, including business analysts, process participants, and technical developers.

The key elements of BPMN 2.0 are:

- Events: Represent occurrences that affect the flow of the process (e.g., start events, end events, intermediate events).
- Activities: Tasks or work that needs to be performed (e.g., user tasks, service tasks, manual tasks).
- Gateways: Decision points that control the divergence and convergence of the process flow (e.g., exclusive gateways, parallel gateways, inclusive gateways).
- Pools and Lanes: Represent major participants in the process, with pools representing different organizations or entities, and lanes representing subdivisions within those entities.
- Artifacts: Supplementary elements such as data objects, groups, and annotations that provide additional information about the process.

In this project, we have decided to utilize BPMN to assist the restaurant manager in understanding the customer ordering process. Specifically, the BPMN diagram will consider both allergies and dietary preferences. Customers will be required to declare any allergies and indicate if they are vegetarian, vegan, or carnivorous. Subsequently, they can decide on the types of meals they would like to order. This approach ensures that the ordering process is tailored to individual needs, enhancing customer satisfaction and safety.

4.2.1 Our BPMN 2.0 Model

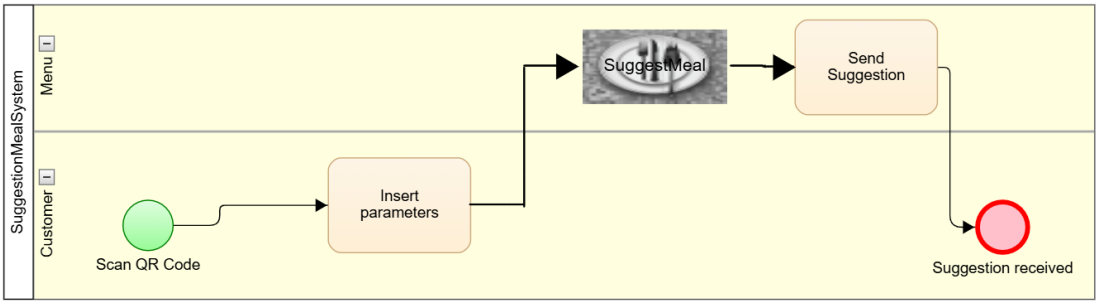

Figure 4.1: BPMN 2.0 graph built in AOAME

As we can see, we have inserted some particular elements to complete the graph. In particular we used a pool element to specify the menu and customer that interact during the whole process. In other words, the customer scan the QR code to see the menu, than he must insert the parameter like allergies and food category, finally the tool provides the suggestions about meals. By implementing this task, we had the opportunity

to work with AOAME (Agile and Ontology-Aided Modeling Environment) which is a methodology and toolset designed to enhance software development and business process modeling by combining agile principles with ontology-based techniques.

After building the BPMN 2.0 we used Jena Fuseki to create some query for examining the graph.
Apache Jena Fuseki is a robust and scalable SPARQL server designed to handle RDF data efficiently. It provides a RESTful interface to perform various operations on RDF datasets, including: Execute SPARQL queries to retrieve data; Perform SPARQL updates to modify data; Load, store, and manage RDF datasets.

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mod: <http://fhnw.ch/modelingEnvironment/ModelOntology#>
PREFIX lo: <http://fhnw.ch/modelingEnvironment/LanguageOntology#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX kebi: <http://www.semanticweb.org/francesco.chiocchi-nicolo.rossini/kebi2024#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT DISTINCT ?property ?value
WHERE {
  mod:SuggestMeal_b8f9fb3d-e3c6-4224-9b85-057a7e4c59a2 ?property ?value .
}
```

Figure 4.2: Query 1: used for see the property and their relative value of our extended class SuggestMeal.

```
SELECT DISTINCT ?meal ?category ?levelCC ?allergy ?course
WHERE {
  mod:SuggestMeal_b8f9fb3d-e3c6-4224-9b85-057a7e4c59a2 lo:meal_hasCategory ?category .
  mod:SuggestMeal_b8f9fb3d-e3c6-4224-9b85-057a7e4c59a2 lo:meal_hasCalorieConsciousLevel ?levelCC .
  mod:SuggestMeal_b8f9fb3d-e3c6-4224-9b85-057a7e4c59a2 lo:meal_containsAllergy ?allergy .
  mod:SuggestMeal_b8f9fb3d-e3c6-4224-9b85-057a7e4c59a2 lo:meal_hasCourse ?course .

  BIND(IF(?category = "Vegetarian", kebi:category_vegetarian,
          IF(?category = "Carnivorous", kebi:category_carnivorous, ?category)) AS ?finalCategory) .

  BIND(IF(?course = "Appetizer", kebi:course_appetizer,
          IF(?course = "First Dish", kebi:course_first-dish,
             IF(?course = "Main Dish", kebi:course_main-dish,
                IF(?course = "Second Course", kebi:course_second-course,
                   IF(?course = "Sidedish", kebi:course_sidedish,
                      IF(?course = "Dessert", kebi:course_dessert, ?course)))))) AS ?finalCourse) .

  BIND(IF(?allergy = "Gluten", kebi:grain_containsGlutenIntolerance,
             IF(?allergy = "Lactose", kebi:dairy_containsLactoseIntolerance, ?allergy)) AS ?finalAllergy) .

  ?meal a kebi:Meal .
  ?meal kebi:meal_hasLevelOfCalorieConscious ?kcal .
  FILTER(?kcal = ?levelCC)

  ?meal kebi:meal_hasCategory ?finalCategory .
  ?meal kebi:meal_hasCourse ?finalCourse .

  OPTIONAL {
    ?meal kebi:meal_hasIngredient ?ingredient .
    ?ingredient rdf:type ?ingredientType .
    FILTER ((?finalAllergy = kebi:grain_containsGlutenIntolerance && ?ingredientType = kebi:Grain) ||
            (?finalAllergy = kebi:dairy_containsLactoseIntolerance && ?ingredientType = kebi:Dairy))
  }
  FILTER (!BOUND(?ingredient))
}
```

Figure 4.3: Query 2: used for analyze the extended class and return the meals which satisfied all data property described.

The first query retrieves the properties and their relative value of SuggestMeal class, while the second one is used to analyze the graph and it provides the meal suggestions that satisfy the customer's parameters.

For the first example, as we can see in the image above, we set up the attributes of our BPMN class and we fire the query in Jena Fuseki that gives back the result.

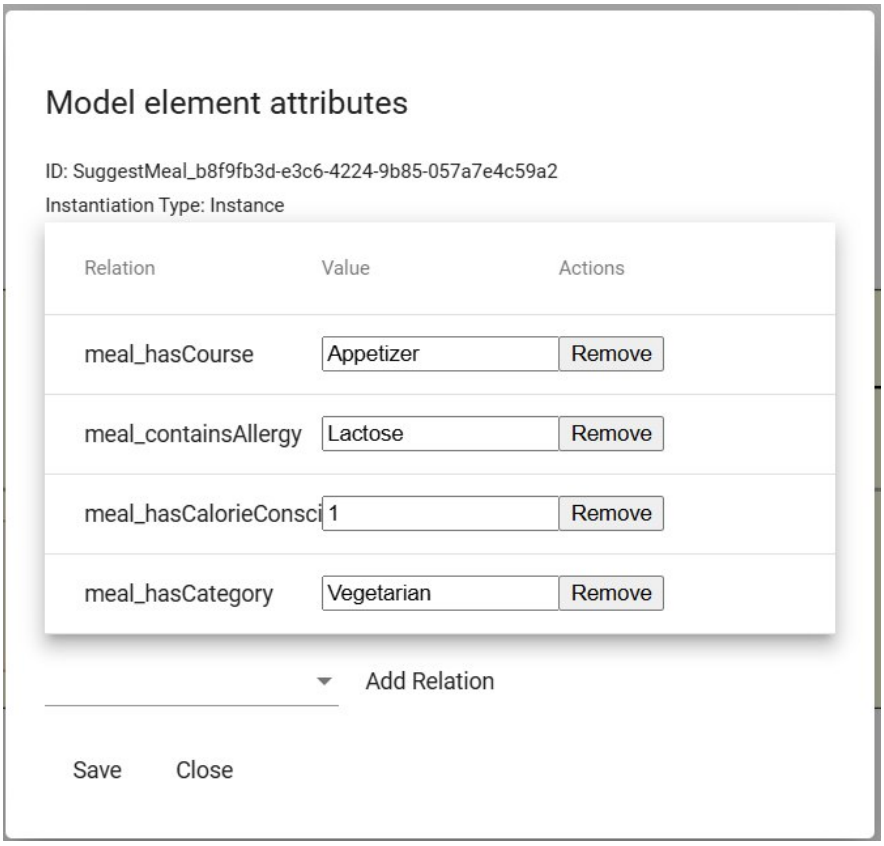

Figure 4.4: Define the attributes of our extended class. We set up the lactose allergy with vegetarian category.

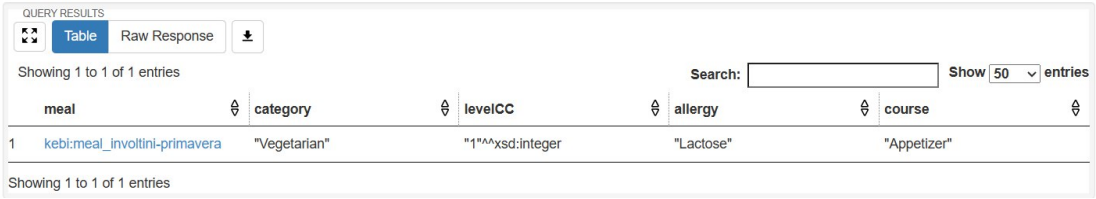

Figure 4.5: Result of query 2 with parameter of figure 4.4

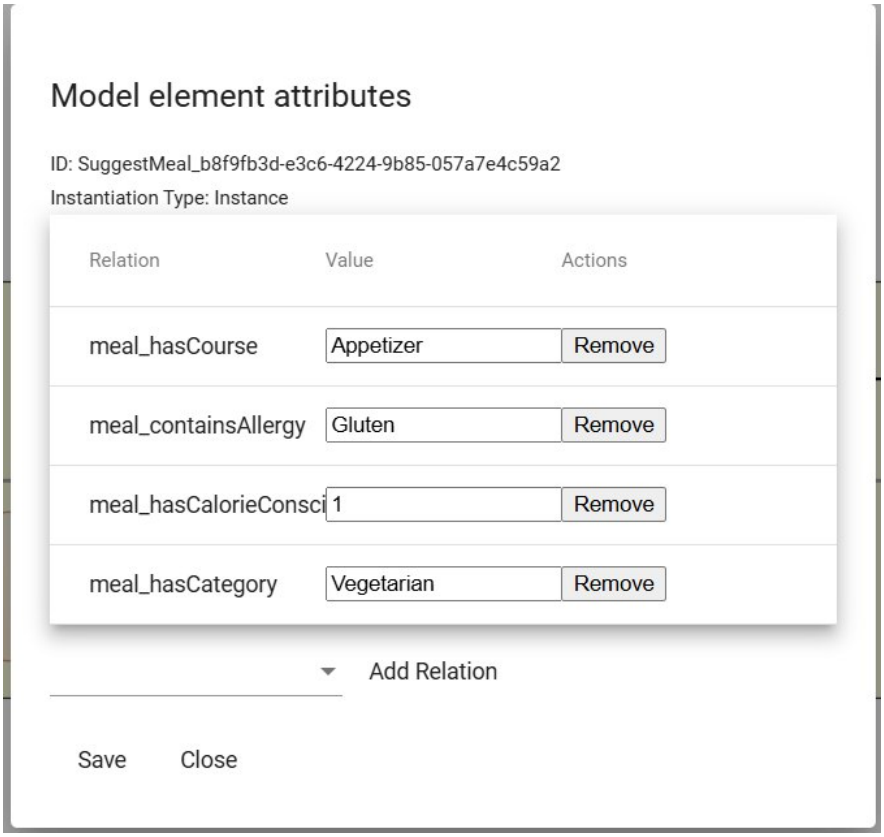

Figure 4.6: Define the attributes of our extended class. We set up the gluten allergy with vegetarian category.

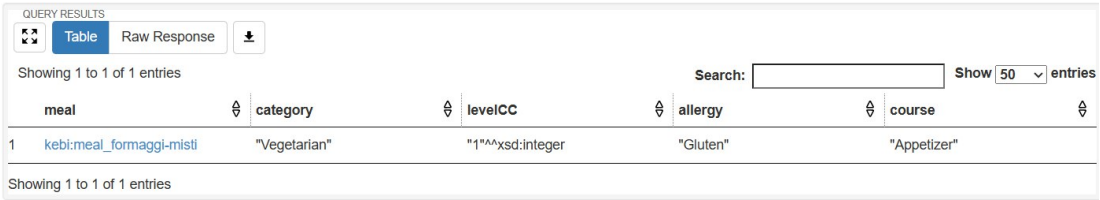

Figure 4.7: Result of query 2 with parameter of figure 4.6

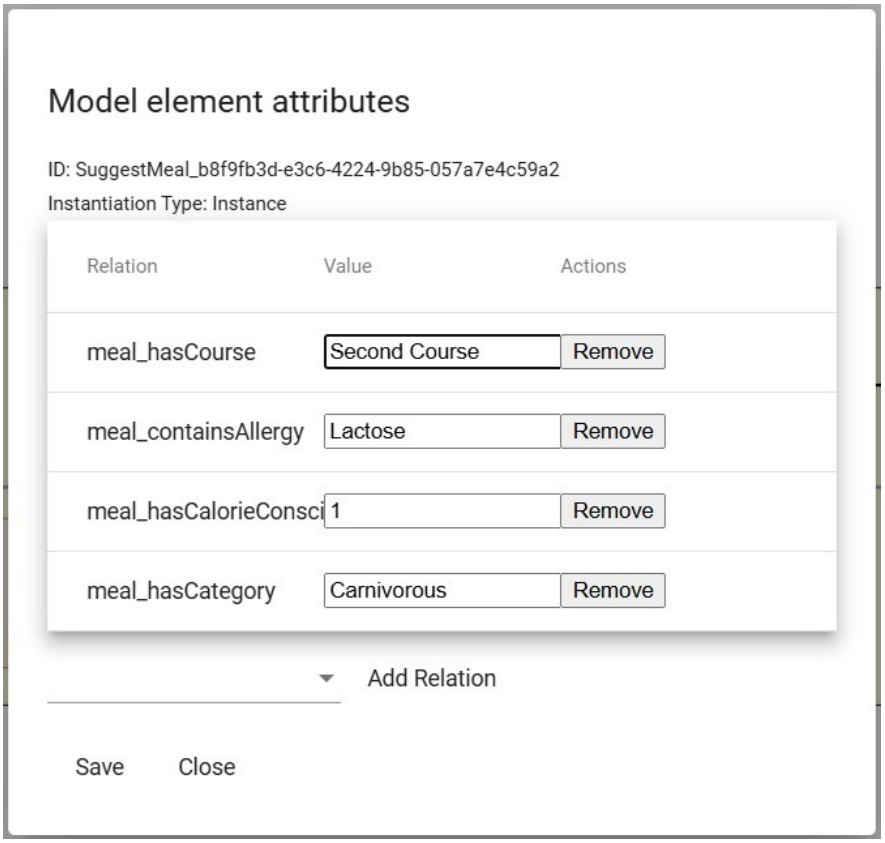

Figure 4.8: Define the attributes of our extended class. We set up the lactose allergy with carnivorous category.

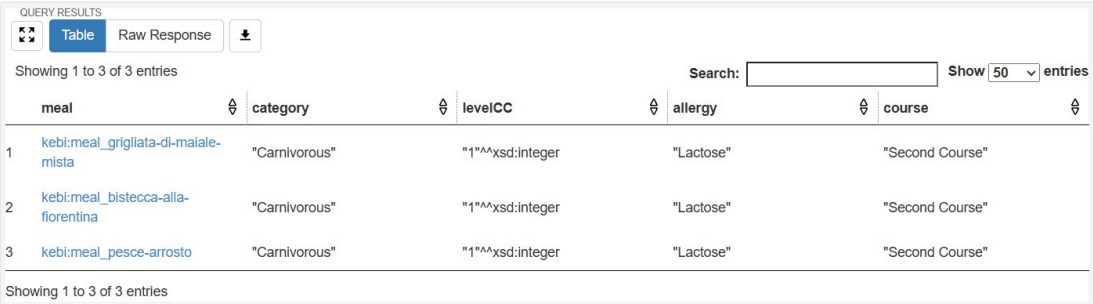

Figure 4.9: Result of query 2 with parameter of figure 4.8

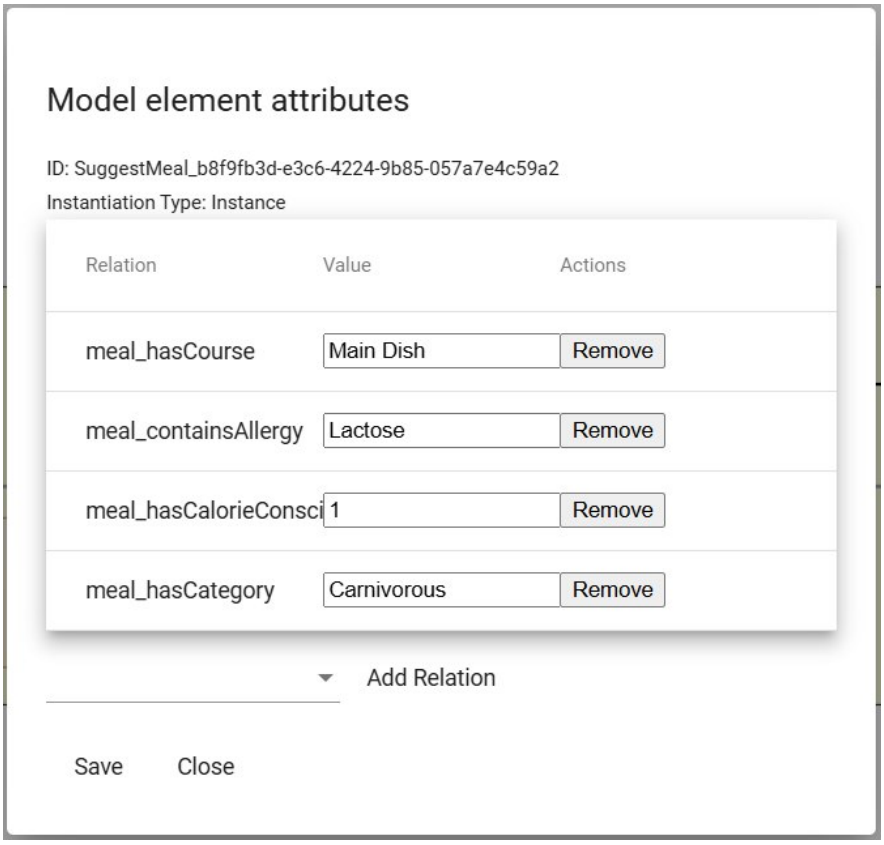

Figure 4.10: Define the attributes of our extended class. We set up the lactose allergy with carnivorous category and main dish.

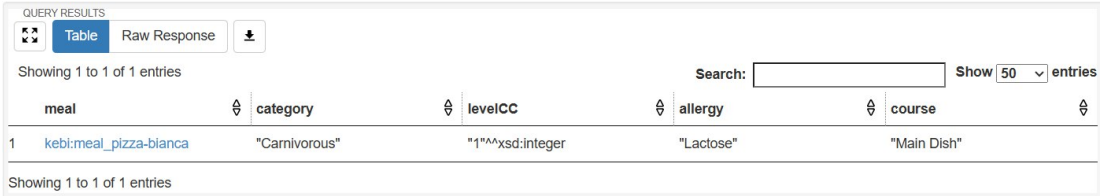

Figure 4.11: Result of query 2 with parameter of figure 4.10

5. Conclusions

5.1 Francesco Chiocchi

During this project, my colleague and I engaged extensively in knowledge engineering, exploring various knowledge-based solutions, each with its own set of advantages and disadvantages. I will now provide an overview of my impressions on these solutions, highlighting both positive and negative aspects.

- **Decision Tables**: Decision tables operate straightforwardly, taking inputs and generating outputs based on predefined rules organized in rows. The selection of outputs follows a "Hit Policy," which dictates how results are grouped. Our project employed a "Collect" policy, enabling concurrent evaluation of all rules, maximizing the potential of Decision Model and Notation (DMN). Under this policy, input conditions are structured in a simple format where columns are linked by an "and" relationship, while rows operate under an "or" relationship with each other. Output results are aggregated into sets for respective output columns. This operational clarity makes it user-friendly and straightforward to implement. However, in complex contexts with numerous variables and intricate conditions, the applicability of decision tables may become impractical or exceedingly intricate. Manipulating data using decision tables also proved to be cumbersome and repetitive. The manual creation of entries became monotonous and tedious, compounded by difficulties in debugging, exacerbated by limited support from available online tools.

- **Prolog**: I thoroughly enjoyed working with Prolog during this project. It provided me with an opportunity to explore a new programming language and conduct extensive experimentation. Prolog, although not excessively challenging to grasp, presents difficulties in accessing comprehensive learning resources. It emphasizes recursion heavily, lacks conventional loop structures, and straightforward arithmetic operations. However, it excels in scenarios where managing data and retrieving information from knowledge bases are straightforward and intuitive. Despite the enjoyable learning experience, Prolog has significant drawbacks. Chief among these is the lack of clarity in some areas. Additionally, while Prolog is optimized for logical operations, handling large knowledge bases can strain its interpreter, resulting in extended computation times for obtaining results.

- **Protégé**: I viewed this approach as a hybrid between DMN (Decision Model and Notation) and Object-Oriented programming languages. In this framework, we could define objects along with their relationships and attributes, and extract knowledge from these object instances. This method of modeling knowledge intrigued me, particularly due to its querying method closely resembling standard SQL, which simplified its usage. I found the SHACL validator highly useful for

testing whether a property is correctly defined or not, likening it to performing assertTrue or assertFalse in Java. SWRL rules enable type inference, but in my view, they should be used sparingly and judiciously. This perspective arises from experiences with querying tools like AOAME or GraphDB, which lack built-in reasoning capabilities for type inference. As a result, queries needed to be meticulously crafted, especially due to rules defined for meals, where inference determined whether a meal contained allergenic ingredients based on its constituents.

- **AOAME**: The Agile Ontology-based Approach for Modeling integrates meta-modeling and modeling to foster effective collaboration between domain experts and language engineers. By leveraging ontologies, it manages knowledge and relationships within the model, enhancing data integration and reasoning capabilities. This approach supports extension, modification, hiding, and deletion of modeling constructs, enabling precise customization of Domain-Specific Modeling Languages (DSMLs). Furthermore, it ensures consistency between human-interpretable and machine-interpretable knowledge through meta-modeling operators, underpinned by SPARQL queries and a triplestore. This alignment supports coherent representations across both human and machine contexts.
  However, AOAME faces challenges such as significant bugs impacting user experience and system stability. Additionally, its online version may struggle during periods of high user traffic, potentially limiting operational efficiency. These issues highlight areas needing improvement to enhance overall reliability and performance.

5.2 Nicolò Rossini

The project aimed to enhance the dining experience by creating a system that customizes restaurant menus according to guest preferences and dietary restrictions. This initiative addresses the challenge of navigating large digital menus on small screens and ensures guests are presented with only the meals they can or prefer to consume.
We developed multiple knowledge-based solutions to recommend meals based on guest profiles using various representation languages:

- Decision Tables: We constructed decision tables to model the decision-making process for meal recommendations. These decision tables encapsulate the logic for filtering meals according to dietary preferences and restrictions, ensuring a systematic and clear approach to decision management.

- Knowledge Graph/Ontology: We created an ontology to represent the detailed knowledge about meals, ingredients, and guest preferences. Using Semantic Web Rule Language (SWRL) for rules and SPARQL for queries. SHACL shapes were employed to validate data consistency. This approach facilitated a rich and extensible framework for capturing and utilizing domain knowledge.

Building on our ontology, we adapted BPMN 2.0 to suggest meals tailored to individual guests. We created a graphical notation that is intuitive for restaurant managers. This notation helps visualize the decision process in selecting appropriate meals based on guest profiles. Using Apache Jena Fuseki, we integrated our ontology into a triple store and executed SPARQL queries to dynamically generate personalized menu suggestions.

The system we developed has significant potential to improve guest satisfaction by providing personalized dining experiences. It also streamlines the decision-making process for restaurant managers, ensuring that guests are only shown meals that meet their dietary needs and preferences.
In conclusion, our project successfully demonstrates the integration of advanced knowledge representation techniques with practical applications in the restaurant industry, paving the way for more intelligent and user-friendly digital menu systems.