UNIVERSITÀ DI CAMERINO

University
University of Camerino

School of Science and Technology
Master Degree in Computer Science
Knowledge Engineering Project

Project "Personalized Menu"

Group members

Tommaso Catervi - tommaso.catervi@studenti.unicam.it
Francesco Santarelli - francesc.santarelli@studenti.unicam.it

Professors
Prof. Knut Hinkelmann
Prof. Holger Wache

Academic Year 2021/2022

Abstract

Knowledge engineering is a field of artificial intelligence (AI) that creates rules to apply to data to imitate the thought process of a human expert. It looks at the structure of a task or a decision to identify how a conclusion is reached.

A library of problem-solving methods and the collateral knowledge used for each can then be created and served up as problems to be diagnosed by the system. The resulting software could then assist in diagnosis, trouble-shooting, and solving issues either on its own or in a support role to a human agent.

Knowledge engineering sought to transfer the expertise of problem-solving human experts into a program that could take in the same data and come to the same conclusion. This approach is referred to as the transfer process, and it dominated early knowledge engineering attempts.

With this in mind, this project's main scope is to define some knowledge-based solutions for an Italian restaurant to build a personalized menu for every kind of preference of the guests. For example if there is a vegan guest the system will show him only dishes where all the ingredients are vegan.

Contents

Introduction

This introduction chapter will speak about the given Project Description and the Scope of the project itself.

1.1 Project Description

With COVID-19 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 The Scope

The actual Scope of the project is the definition of three different knowledge-based solution based on Decision Tables in Camunda Modeler, Prolog and Knowledge Graph in Protege.

To do that it is needed to study the theory of Knowledge Engineering and to practice with the different knowledge-based languages and editors.

Technologies

The purpose of this chapter is presenting the technologies and frameworks studied and used during the realization of the project, briefly introducing their functionalities and employment.

2.1 Camunda Modeler

The **Camunda platform** (Figure 2.1) provides an innovating process automation with a standards-based, highly scalable and collaborative approach for business and IT.

Processes are the algorithms that determine how an organization runs. Successful businesses grow from proven, effective processes.

The Camunda's mission is to enable organizations to design, automate and improve these processes, no matter where they are and what they entail.

Figure 2.1: Camunda Logo.

To create the Decision Tables for this project, we opted for the **Camunda Modeler** (Figure 2.2), which is a desktop application that can be installed and used locally, all while integrating your local development environment.

Figure 2.2: Camunda Modeler Icon.

We tested the results of our Decision Tables with the online tool Camunda DMN-Simulator, that can simulate the output of the DMN, showing errors in some cases and tips to correct them.

2.2 Prolog

Prolog is a logic programming language (Figure 2.3). It has important role in artificial intelligence.

Figure 2.3: Prolog Icon.

Unlike many other programming languages, Prolog is intended primarily as a declarative programming language. In prolog, logic is expressed as relations (called as Facts and Rules).

Core heart of prolog lies at the logic being applied. Formulation or Computation is carried out by running a query over these relations.

In prolog, We declare some facts. These facts constitute the Knowledge Base of the system. We can query against the Knowledge Base. We get output as affirmative if our query is already in the knowledge Base or it is implied by Knowledge Base, otherwise we get output as negative.

So, Knowledge Base can be considered similar to database, against which we can query. Prolog facts are expressed in definite pattern. Facts contain entities and their relation.

2.3 Online Prolog editor

SWI-Prolog is a versatile implementation of the Prolog language (Figure 2.4).

Figure 2.4: Online Prolog editor Logo.

Although SWI-Prolog gained its popularity primarily in education, its development is mostly driven by the needs for application development.

This is facilitated by a rich interface to other IT components by supporting many document types and (network) protocols as well as a comprehensive low-level interface to C that is the basis for high-level interfaces to C++, Java (bundled), C#, Python, Rust, etc. (externally available).

SWI-Prolog aims at scalability. It unifies many extensions of the core language that have been developed in the Prolog community such as tabling, constraints, global variables, destructive assignment, delimited continuations and interactors.

2.4 Visual Studio Code

Visual Studio Code (Figure 2.5) is a lightweight but powerful source code editor which runs on your desktop and is available for Windows, macOS and Linux.

Figure 2.5: Visual Studio Code Logo.

Visual Studio Code comes with built-in support for JavaScript, TypeScript and Node.js and has a rich ecosystem of extensions for other languages and runtimes (such as C++, C#, Java, Python, PHP, Go, .NET).

It enables additional languages, themes, debuggers, commands, and more thanks to its Extensions. VS Code's growing community shares their secret sauce to improve your workflow.

We used Visual Studio Code to write the prolog facts and rules using an extension that enabled us to work with ".pl" file type.

2.5 Protege

Protege (Figure 2.6) is a free, open-source ontology editor and framework for building intelligent systems.

Figure 2.6: Protege Logo.

Protégé is supported by a strong community of academic, government, and corporate users, who use Protégé to build knowledge-based solutions in areas as diverse as biomedicine, e-commerce, and organizational modeling.

Its plug-in architecture can be adapted to build both simple and complex ontology-based applications.

Developers can integrate its output with rule systems or other problem solvers to construct a wide range of intelligent systems. Most important, the Stanford team and the vast Protégé community are here to help.

Decision Tables

This chapter will illustrate the development of a knowledge-based solution based on DMN and decision tables.

3.1 Decision Model and Notation

DMN is a modeling language and notation for the precise specification of business decisions and business rules. DMN is easily readable by the different types of people involved in decision management. These include: business people who specify the rules and monitor their application; business analysts.

DMN is designed to work alongside BPMN and/or CMMN, providing a mechanism to model the decision-making associated with processes and cases. While BPMN, CMMN and DMN can be used independently, they were carefully designed to be complementary. Indeed, many organizations require a combination of process models for their prescriptive workflows, case models for their reactive activities, and decision models for their more complex, multi-criteria business rules. Those organizations will benefit from using the three standards in combination, selecting which one is most appropriate to each type of activity modeling. This is why BPMN, CMMN and DMN really constitute the “triple crown” of process improvement standards.

3.2 The diagram

We tried to imagine how a DMN diagram could be implemented in a realistic scenario of a hypothetical application through which a restaurant customer could view a menu customized to their needs, and we concluded how the interface of such an application could consist of a series of buttons that could allow the user to apply or remove filters to the menu (such as "gluten-free" or "vegan") and a final button to generate the filtered menu based on the filters applied.

In the latter operation, the front-end of the application would certainly dialogue with a back-end (via REST calls for example) asking for a list of all dishes that meet the constraints imposed by the user. For this reason, we envisioned user preferences as possible boolean flags that could be cascaded in order to have as many scenarios as possible to satisfy even the most demanding client.

In this way, the user's choices can be simplified to trivial "yes" or "no" regarding the application of a given filter, such as:

- "Are you interested in vegan dishes?" YES or NO
- "Are you interested in lactose-free dishes?" YES or NO
- and so on...

Eventually, the back-end would make a hypothetical query to a database asking for the list of dishes that satisfy all and only the flags marked as "true" ("YES").

However, to know whether or not a dish corresponds to the needs of a specific customer, it is necessary to go back to the list of its ingredients and find out whether it contains foods that

the customer cannot eat. Therefore, in our idea, the starting point was to identify the list of all ingredients that the customer can eat based on the value of the flags and then, only then, to calculate the list of dishes that exclusively contain those ingredients.

Our final diagram (Figure 3.1) in fact consists of two tables, "ingredients" and "meals."

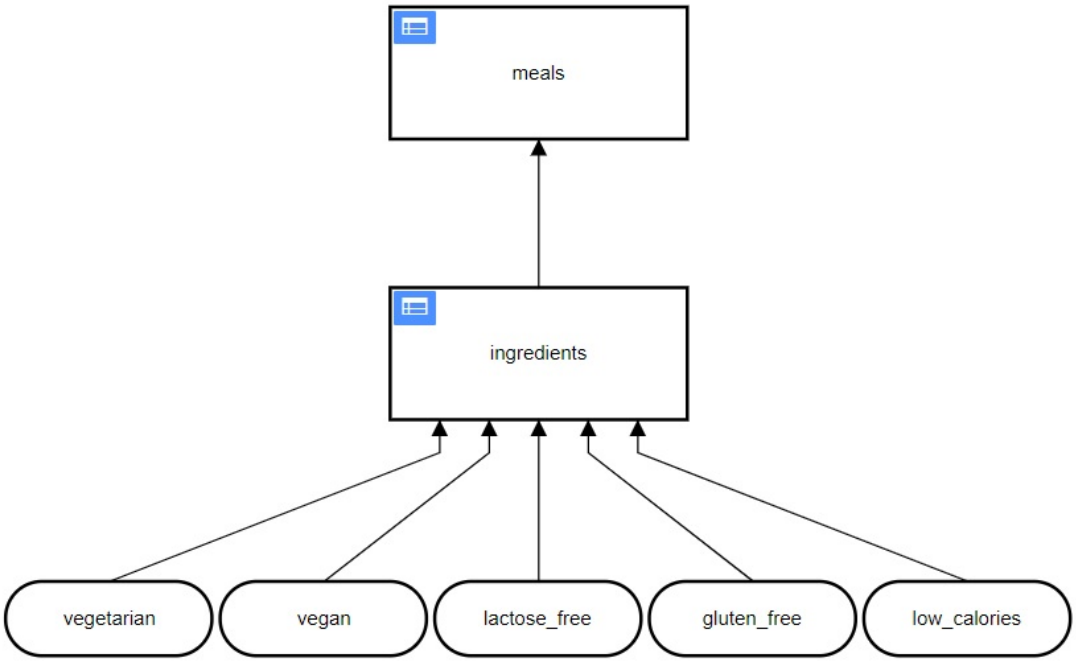
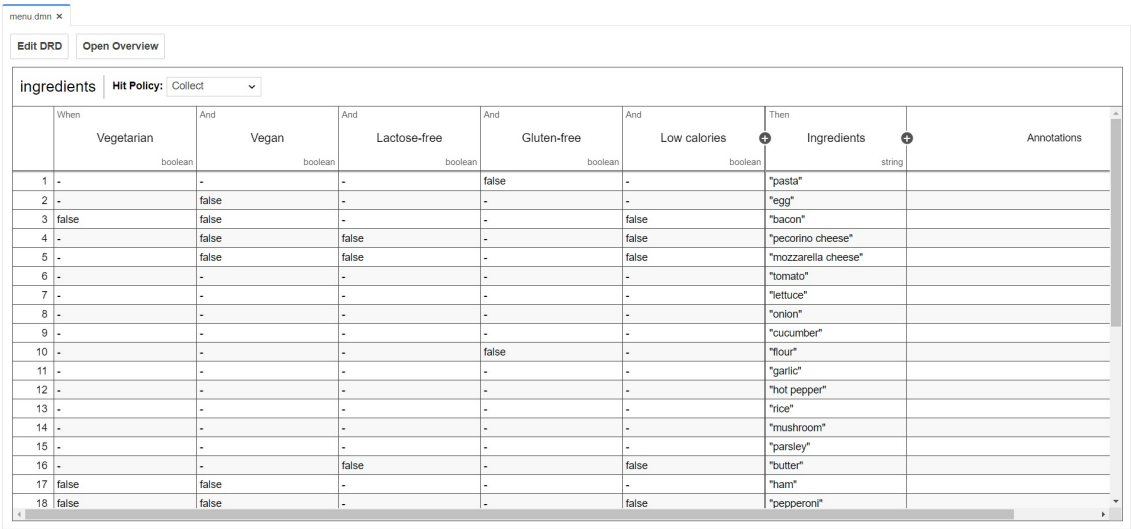


Figure 3.1: DMN diagram for the personalized menu.

3.2.1 The "ingredients" table



menu.dmn ×

Edit DRD | Open Overview

ingredients | Hit Policy: Collect

	When Vegetarian boolean	And Vegan boolean	And Lactose-free boolean	And Gluten-free boolean	And Low calories boolean	Then Ingredients string	Annotations
1	-	-	-	false	-	"pasta"	
2	-	false	-	-	-	"egg"	
3	false	false	-	-	false	"bacon"	
4	-	false	false	-	false	"pecorino cheese"	
5	-	false	false	-	false	"mozzarella cheese"	
6	-	-	-	-	-	"tomato"	
7	-	-	-	-	-	"lettuce"	
8	-	-	-	-	-	"onion"	
9	-	-	-	-	-	"cucumber"	
10	-	-	-	false	-	"flour"	
11	-	-	-	-	-	"garlic"	
12	-	-	-	-	-	"hot pepper"	
13	-	-	-	-	-	"rice"	
14	-	-	-	-	-	"mushroom"	
15	-	-	-	-	-	"parsley"	
16	-	-	false	-	false	"butter"	
17	false	false	-	-	-	"ham"	
18	false	false	-	-	false	"pepperoni"	

Figure 3.2: Ingredients table.

The "ingredients" table (Figure 3.2) takes as input five boolean parameters, which are respec-

tively:

- **vegetarian**
- **vegan**
- **lactose_free**
- **gluten_free**
- **low_calories**

and returns as output the name of the single ingredient as a string. By applying the hit policy "collect" we then got the list of all ingredients. With hit policy collect, you do not care about the order or any interdependencies between your rules at all. Instead, you just "collect" independent rules and care about the question which rules are applicable to your specific case.

The output of the "ingredients" table is then passed as the only input to the "meals" table.

3.2.2 The "meals" table

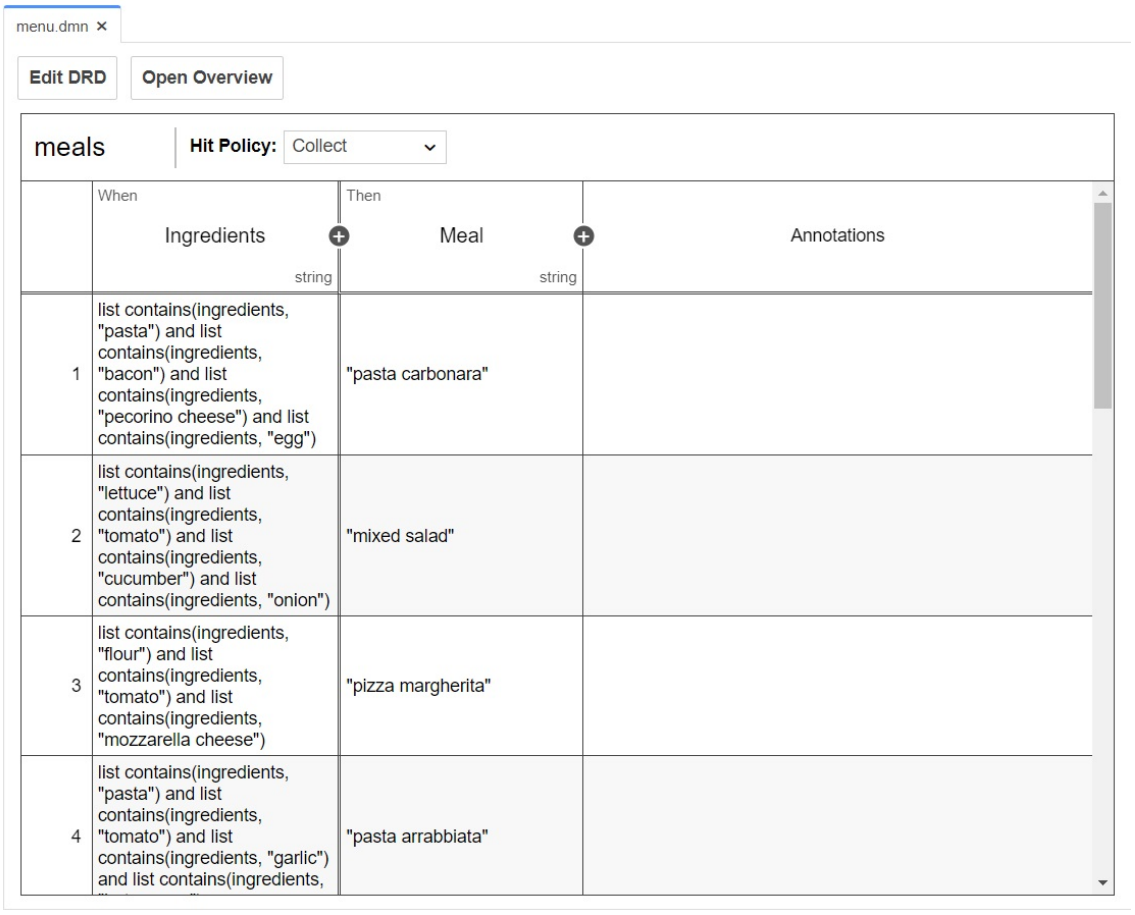


Figure 3.3: Meals table.

The "meals" table (Figure 3.3) takes as input a single string parameter (i. e. the list returned from the previous table), which is:

- **ingredients**

and returns as a string output the name of the individual dish that has as ingredients only those contained in the list passed as input. Again, the hit policy "collect" was used to group all the dishes together (Figure 3.4).

The "list contains" function belonging to the FEEL language was used to check that the list of input ingredients contains particular values.

FEEL (Friendly Enough Expression Language) is a part of the DMN specification of the Object Management Group (OMG). It is designed to write expressions for decision tables and literal expressions in a way that is easily understood by business professionals and developers.

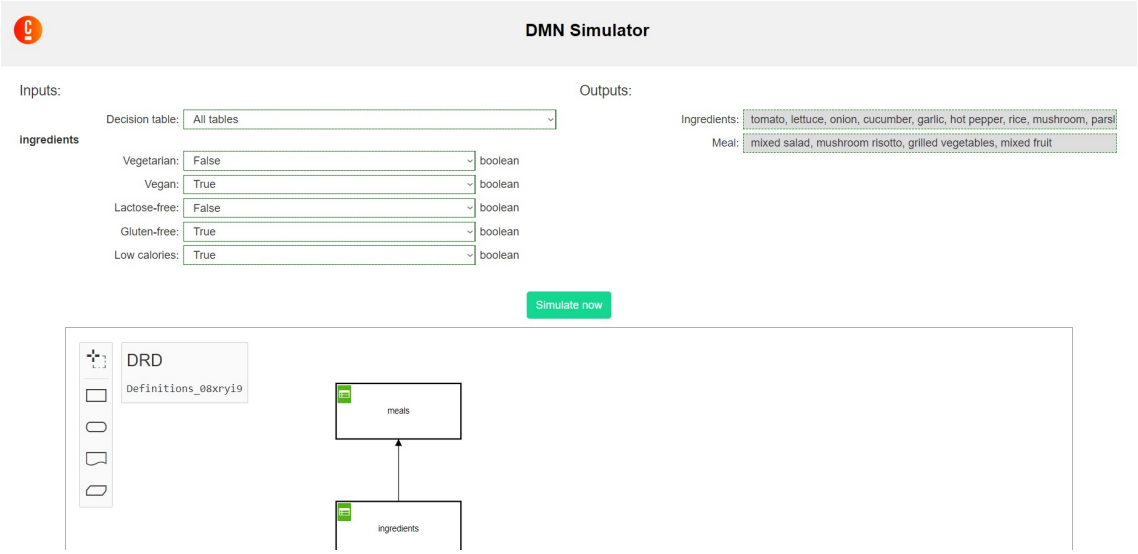


Figure 3.4: Simulation of the final DMN diagram.

Prolog

This chapter will illustrate the development of a knowledge-based solution based on Prolog.

4.1 Facts and Rules

To model a knowledge-based solution with Prolog, we followed the same idea adopted for decision tables, which is to describe the ingredients in terms of characteristics (i. e. "lactose-free", "vegan", etc)

To do so, we defined a set of facts to describe the ingredients and dishes.

A fact is a predicate expression that makes a declarative statement about the problem domain. Whenever a variable occurs in a Prolog expression, it is assumed to be universally quantified.

A rule in Prolog is a clause, normally with one or more variables in it. Normally, rules have a head, neck and body, as in: eats(Person, Thing) :- likes(Person, Thing), food(Thing). This says that a Person eats a Thing if the Person likes the Thing , and the Thing is food.

The facts include:

- vegetarian(egg).
- vegan(rocket_salad).
- carnivore(bacon).
- has_lactose(pecorino_cheese).
- has_gluten(breadcrumbs).
- has_low_calories(tomato).
- meal(pasta_carbonara).
- ingredient(pasta_carbonara, bacon).

4.2 User interface

In order to test the knowledge-based solution with Prolog, we made use of the online SWI-Prolog tool. To continue with the testing you have to paste your facts and rules into SWI-Prolog and it enables you to execute your rules and give you an output to test your solution.

In order to provide the user with an easier way to enter preferences, we then implemented a choice menu (Figure 4.1) that asks the user to select various preferences by entering a value between 1 and 2 according to the choice.

Once the user's preferences are obtained, the list of dishes is filtered for each constraint

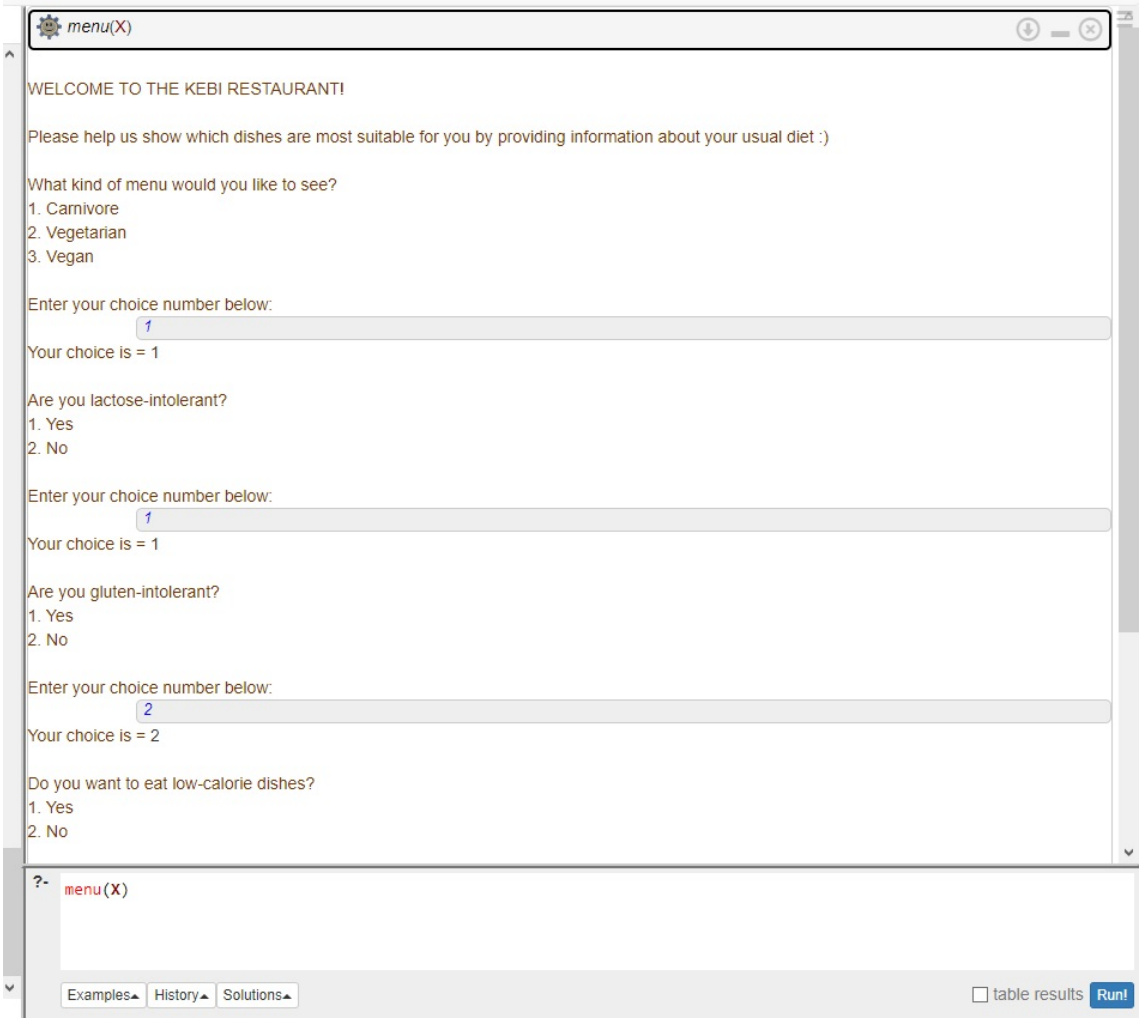


Figure 4.1: Final choice menu in prolog.

To bring up the choice menu, simply execute the rule:

- **menu(X).**

where X corresponds to a dish. The end result (Figure 4.2) will be the list of all possible values of X, i.e. the list of dishes corresponding to the requirements specified by the customer

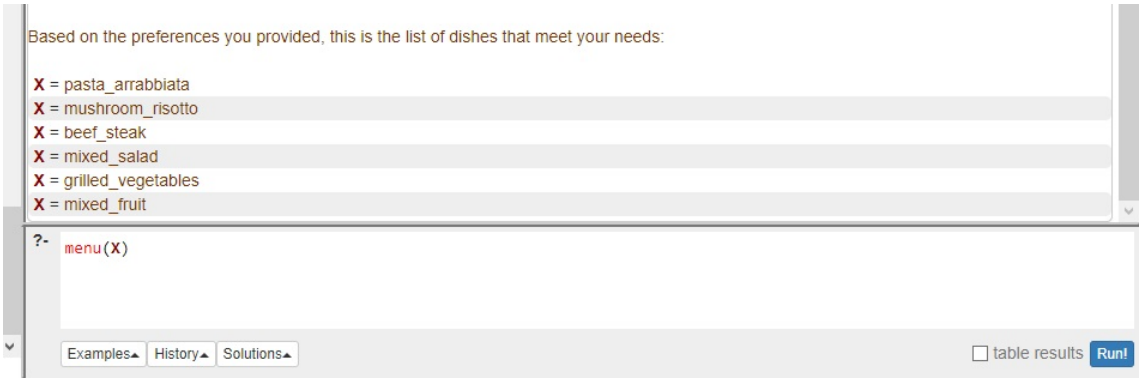


Figure 4.2: Final result with the list of dishes.

Knowledge Graphs

This chapter will illustrate the development of an Ontology Structure using the editor Protege and the creation of a Knowledge Graph.

5.1 Ontology on Protege

To create the ontology structure for our project we used the **Protege Editor**, that is free and open-source.

5.1.1 Class Hierarchy

The **Class Hierarchy** view displays the asserted and inferred class hierarchies (Figure 5.1). The asserted class hierarchy is visible by default.

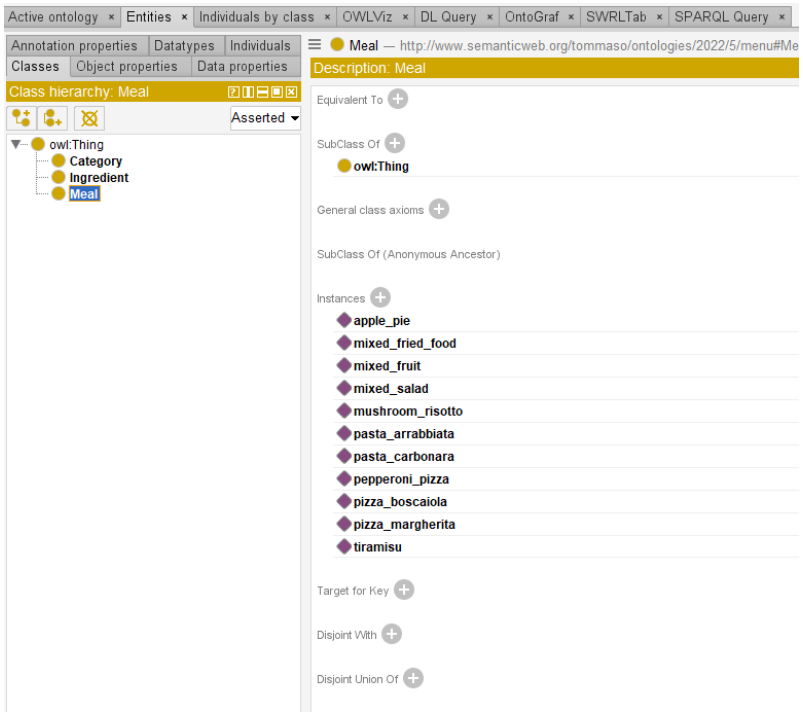


Figure 5.1: Class Hierarchy.

The asserted class hierarchy view is one of the primary navigation devices in Protégé. It is presented as a tree where nodes in the tree represent classes.

A child-parent relationship in the tree represents a sub/super class relationship in the class hierarchy.

In our case we have three classes: Category, Meal and Ingredient. All of them contain a set of individuals created by us to better represent the real scenario cases. For example the class Meal contains the menu of an Italian restaurant, with a full list of dishes.

In fact, Individuals in Protege are typed as members of one or more classes organized and displayed as a hierarchy.

5.1.2 Object Property Hierarchy

The **Object Property Hierarchy** view displays the asserted and inferred object property hierarchies. The asserted object property hierarchy is visible by default.

The Object Properties connect two individuals (a subject and object) with a predicate.

The asserted object property hierarchy view is one of the primary navigation devices in Protégé. It is presented as a tree where tree nodes correspond to object properties.

A child node represents an object property that is a subproperty of the property represented by the parent node.

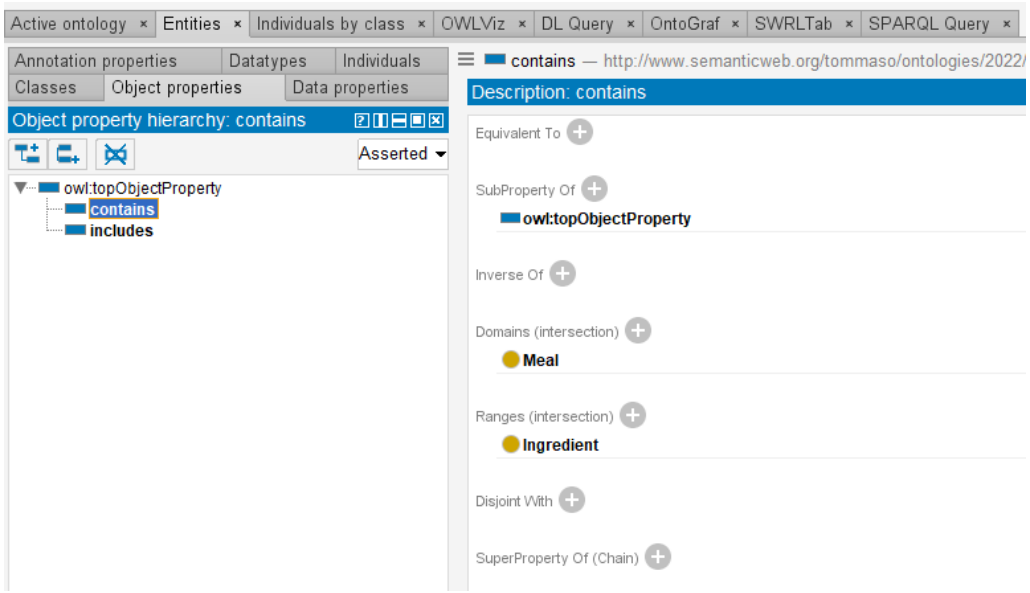


Figure 5.2: Object Properties Hierarchy.

In our case, (Figure 5.2), we have two Object Properties: contains and includes.
The contains property describe the relation with the Meal class as Domain and the Ingredient class as Ranges, it says "a meal contains the ingredient x".
The includes property describe the relation with the Category class as Domain and the Ingredient class as Ranges, it says "a category includes the ingredient x".

5.1.3 Data Property Hierarchy

The **Data Property Hierarchy** view displays the asserted and inferred data property hierarchies. The asserted data property hierarchy is visible by default.

In the Data Properties the predicate connects a single subject with some form of attribute data. Data properties have defined datatypes including string, integer, date, datetime, or boolean.

The asserted data property hierarchy view is one of the primary navigation devices in Protégé. It is presented as a tree where tree nodes correspond to data properties.

A child node represents an data property that is a sub-property of the property represented by the parent node.

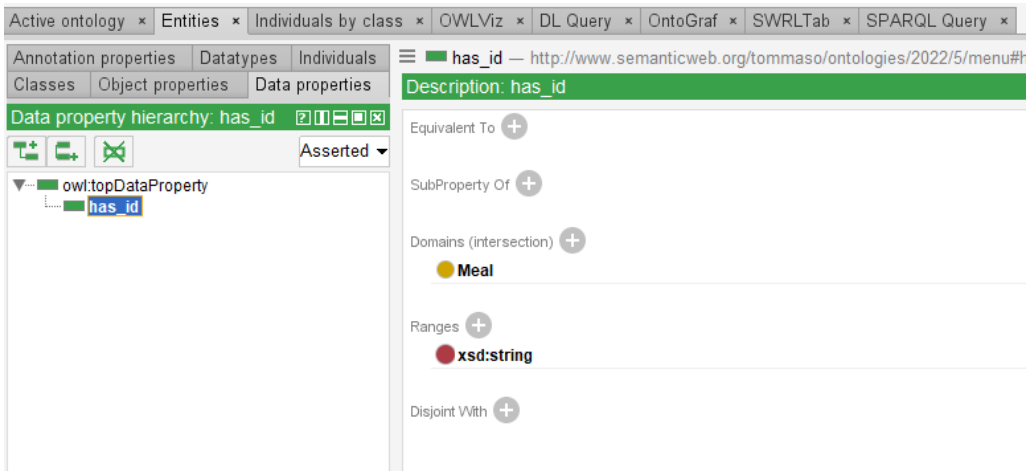


Figure 5.3: Data Properties Hierarchy.

In our case, (Figure 5.2), we have two Object Properties: contains and includes.
The contains property describe the relation with the Meal class as Domain and the Ingredient class as Ranges, it says "a meal contains the ingredient x".
The includes property describe the relation with the Category class as Domain and the Ingredient class as Ranges, it says "a category includes the ingredient x".

5.2 Resulting Knowledge Graph

A **Knowledge Graph**, also known as a semantic network, represents a network of real-world entities—i.e. objects, events, situations, or concepts and illustrates the relationship between them.

This information is usually stored in a graph database and visualized as a graph structure, prompting the term knowledge “graph”.

A knowledge graph is made up of three main components: nodes, edges, and labels. Any object, place, or person can be a node. An edge defines the relationship between the nodes.

The resulting Knowledge Graph (Figure 5.4) is divided into the classes Category, Meal and Ingredient.

As visible in the figure the Category class is connected with the Ingredient class by the Object Property ”Includes” and Meal class is connected with the Ingredient class by the Object Property ”Contains”.

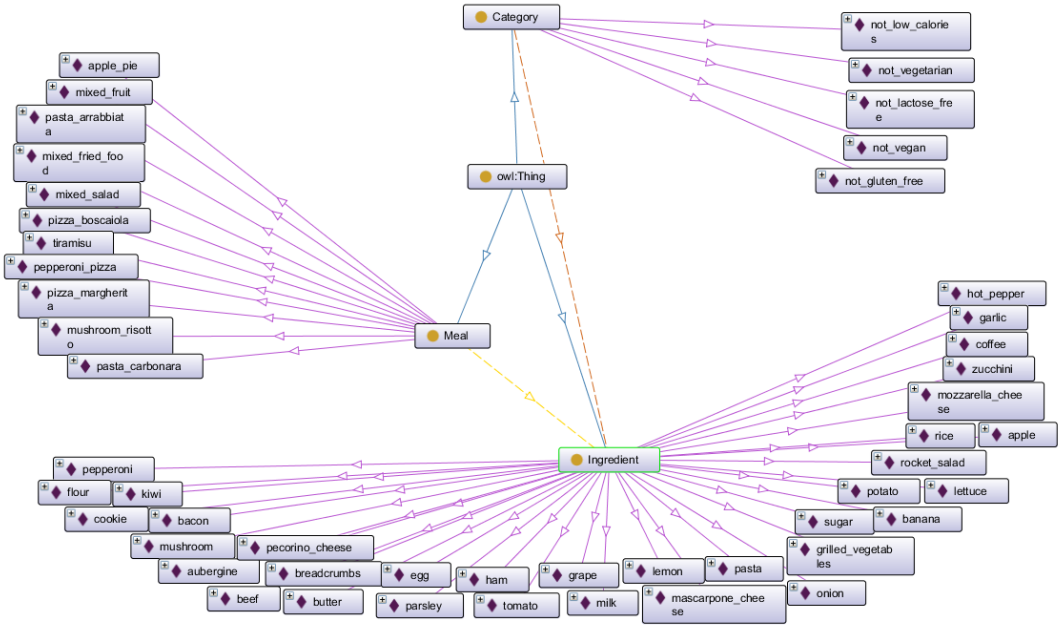


Figure 5.4: Resulting Knowledge Graph.

5.3 SPARQL Query

SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. SPARQL also supports extensible value testing and constraining queries by source RDF graph. The results of SPARQL queries can be results sets or RDF graphs.

In our case, we defined a set of query to get the different filtered menu.

SPARQL query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX po: <http://www.semanticweb.org/tommaso/ontologies/2022/5/menu#>
SELECT ?meal
        WHERE {
                ?meal po:has_id ?id.
                MINUS {
                        ?meal po:contains ?ingredient.
                        po:not_vegan po:includes ?ingredient
                }.
                MINUS {
                        ?meal po:contains ?ingredient.
                        po:not_lactose_free po:includes ?ingredient.
                }.
                MINUS {
                        ?meal po:contains ?ingredient.
                        po:not_gluten_free po:includes ?ingredient.
                }.
                MINUS {
                        ?meal po:contains ?ingredient.
                        po:not_low_calories po:includes ?ingredient.
                }
        }
```

```
mixed_salad
mushroom_risotto
mixed_fruit
grilled_vegetables
```

Figure 5.5: SPARQL Query.

For example, in the (Figure 5.5), we defined a query for the "worst case scenario", when the guest wants to choose a vegan menu, he is gluten-intolerant, lactose-intolerant and wants to eat low-calories dishes.

As visible in the figure the resulting output is the filtered menu with a list of dishes that the specific guest can eat: mixed_salad, mushroom_risotto, mixed_fruit and grilled_vegetables.

Conclusions

The scope of the project was to create different knowledge-based solutions based on:

- Decision tables
- Prolog
- Knowledge graph

Thanks to the work done with my project team we managed to define all the three solutions required.

6.1 Decision Tables

Decision Tables and DMN are strong tools that enable you to express logical relationships in an clear way.

They are mainly used to define in an understandable manner the decision statement of a problem in a strict tabular form.

A decision table is a set of conditions inside a table useful to define a problem and the possible solution to deal with it.

Some advantage on using Decision Tables:

- They are mostly easy to define;
- You have a sort of form to "interact" with the user and get your inputs in a simple way;
- They provide a compact representation of the decision making process. A small table can be replace several pages of facts and rules for example;
- Decision table can be changed according to the situation.

The main disadvantage I encountered is that sometimes you have restricted possibilities on defining rules, you cannot express everything you want just with simple rows and columns.

To overcome this obstacle, we started to study the feel language that enabled us to create a set of more complex logic relationships inside the decision tables and thanks to that we arrived to the final solution.

6.2 Prolog

Prolog is a logic and declarative programming language where the logic is expressed as relations called as Facts (that define the Knowledge-Base of the system) and Rules.

This is the solution I liked the most because Prolog let you a lot of freedom and it enables you to represent all you want to represent.

Some of its advantages, in my opinion, are:

- It helps you to have a strong picture in mind of both the problem and the solution;
- It enables you to represent Facts and Rules in a clear way;
- Useful for problem-solving and understanding of the natural language;
- Possibility to create some sort of simple user interface;
- Its language is not too complex, it's easy to learn.

The disadvantages I saw while using Prolog:

- Its programming language does not support the OR and the NOT conditions, but we overcame this problem using other techniques to define these statements;
- The input and output procedures sometimes are not easy;
- Prolog in general does not support graphics, but in our case we didn't need it;
- You usually need a lot of code to get to the solution.

6.3 Knowledge Graph

The **Knowledge Graph** solution comes from the Protege editor, we defined the Class Hierarchy, the Data Properties and the Object Properties and we arrived to the solution thanks to the SPARQL queries.

The main advantage of the Knowledge Graph solution is the possibility to create and see the graph itself, thanks to some type of tool, you can see the elements and their hierarchy in a clear graphical way.

A disadvantage, in my opinion, of Protegee is that it is an academic open-source ontology editor created by the Stanford University, and to be honest is not easy to find useful material on it, besides the Protege official documentation, in fact we studied on it to achieve the final solution.

Moreover, you can get outputs only from the SPARQL query, while in protege and decision table you can create some sort of "interaction" with the user (with a form or with a simple user interface).

List of Figures