University of Camerino

School of Sciences and Technologies

Master degree in Computer Science (Class LM-18)

Personalized Menu

KEBI - Project

Group members

Samuele Pirani - 131034
Matteo Machella - 131725

Supervisors

Prof. Dr. Knut Hinklemann
Prof. Dr. Emanuele Laurenzi

A.A. 2024/2025

Contents

List of Figures

1. Introduction

The purpose of this document is to describe the entire development process of the project of the course Knowledge Engineering and Business Intelligence. We will discuss about all the tasks to be done, the methodology and technologies used, and a brief project goal description.

In Section 1.1 a brief description of the system to be developed will be reported, while in Section 1.2 all the tasks to be completed for the finalization of the project will be illustrated.

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 cannot or do not want every meal, e.g. vegetarians or guests with an allergy. Instead of showing all the meals 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 to create a system that allows for the selection of meals that match the guest preferences.

The knowledge base shall contain information about typical meals of an Italian restaurant, e.g. pizza, pasta, and main dishes. The meals consist of ingredients. There are different types of ingredients, such as meat, vegetables, fruits, or dairy. For each ingredient, there is information on calories.

Guests can be carnivores, vegetarians, calorie conscious, or suffer from allergies, e.g. lactose or gluten intolerance.

1.2 Task List

The following list describes all the tasks to tackle in order to complete the final project:

1. Create different knowledge-based solutions for recommending food depending on the profile of a guest (carnivores, vegetarians, calorie-conscious, suffering from allergies, etc.) using 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-modeling: 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 modeling element, which could be easy to understand for the restaurant manager. Use the triple store interface (Jena Fuseki) to fire the query result.

2. Knowledge Based Solution

In this chapter, the knowledge-based solution realized for the personalized-menu is presented in detail. As described in Section 1.1, the initial interpretation relied on Decision Tables (see Section 2.1) to filter the restaurant's dishes based on guests' preferences.

After the definition of Decision Tables, we provided an implementation of the Prolog script to create the recommendation system for the personalized menu (see Section 2.2.

2.1 Decision Tables

The first phase of the solution is built upon a Knowledge Source, referred to as the *Restaurant Source*. This source feeds the initial decision, called *Menu*, within which the complete dish list of the restaurant is defined.


```
{ "name": "Tomato Bruschetta",
  "ingredients": [
    { "name": "Bread", "type": "Cereal","calories": 250 },
    { "name": "Tomato", "type": "Vegetable", "calories": 18 },
    { "name": "Garlic", "type": "Vegetable", "calories": 149 },
    { "name": "Basil", "type": "Vegetable", "calories": 22 }
  ],
  "course": "Appetizer",
  "allergens": ["gluten"]
},
{ "name": "Seafood Risotto",
  "ingredients": [
    { "name": "Rice", "type": "Cereal","calories": 130 },
    { "name": "Shrimp", "type": "Fish", "calories": 85 },
    { "name": "Squid", "type": "Fish", "calories": 92 },
    { "name": "Mussels", "type": "Fish", "calories": 86 },
    { "name": "Garlic", "type": "Vegetable", "calories": 149 },
    { "name": "Parlsey", "type": "Vegetable", "calories": 36 }
  ],
  "course": "First",
  "allergens": ["shellfish", "mollusks"]
},
  { "name": "Saltimbocca alla Romana",
  "ingredients": [
    { "name": "Veal slice", "type": "Meat","calories": 110 },
    { "name": "Prosciutto crudo", "type": "Meat", "calories": 150},
    { "name": "Sage", "type": "Vegetable", "calories": 50 },
    { "name": "Butter", "type": "Dairy", "calories": 325 }
  ],
  "course": "Main",
  "allergens": ["lactose"]
},
 { "name": "Tiramisu",
  "ingredients": [
    { "name": "Lady fingers", "type": "Cereal","calories": 200 },
    { "name": "Eggs", "type": "Dairy", "calories": 143},
    { "name": "Mascarpone cheese", "type": "Dairy", "calories": 210 },
    { "name": "Coffee", "type": "Vegetable", "calories": 2 },
    { "name": "Cocoa powder", "type": "Vegetable", "calories": 138 }
  ],
```


Figure 2.1: Dish List defined inside the Menu decision

Figure 2.1 illustrates this dish list. Once this central decision is established, the system is designed to further refine the menu through specialized decision modules that focus on three main aspects: filtering dishes based on allergenic content, tailoring the selection to match guest dietary preferences, and aligning dishes with caloric requirements. The module that handles allergenic content takes into account common allergens such as lactose, eggs, peanuts, tree nuts, fish, shellfish, mollusks, soy, and gluten. These allergens have been identified as the most significant due to their frequent occurrence and potential to affect a considerable portion of the clientele. In parallel, the solution considers the guest's dietary profile, taking into account preferences that range from carnivorous diets to vegetarian, vegan, and fish-based diets, thereby capturing a broad spectrum of culinary tastes. In addition, the system evaluates the caloric aspect by distinguishing between various nutritional levels. It categorizes dishes in relation to their energy content by defining ranges such as Light or Diet Friendly (200 to 400 kcal), Moderate or Balanced (200 to 700 kcal), Hearty or Energy Rich (200 to 1000 kcal), and High Calorie (above 1000 kcal). This structured approach ensures that a wide range of input data is available for the decision-making process, which ultimately enhances the precision of the personalized recommendations. The implementation of these decision processes is illustrated by several figures.

Figure 2.2 presents the decision table used to filter out dishes containing allergens, ensuring that no item contradicts the specific health or dietary constraints of the guest.

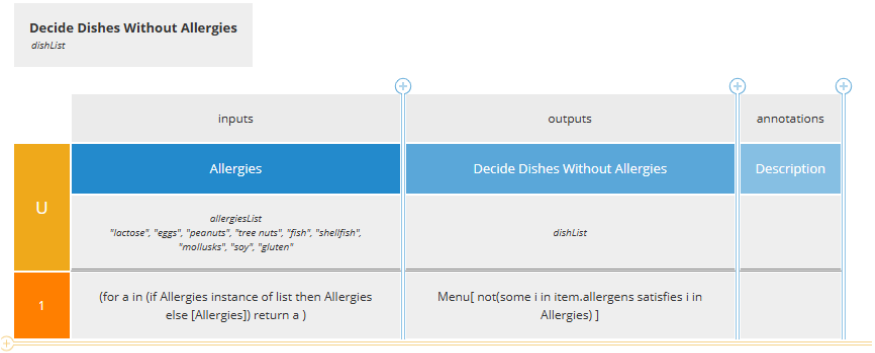

Figure 2.2: Filter Allergies Decision Table

Similarly, Figure 2.3 shows the decision table that matches dishes to the guest's declared profile, and Figure 2.4 details the decision table for aligning the dishes with the guest's caloric profile. These sub-decisions operate sequentially and their outcomes are subsequently merged by a main decision module known as *Filter Remaining Menu*. The purpose of this module is to reconcile the individual outputs of the allergenic, dietary profile, and caloric filters so that a coherent set of options is forwarded to the final decision table, named *Suggest Personalized Menu*.

This table is responsible for showing the final suggestion to the user based on the conditions entered previously. A suggestion consists of two pairs of values named *Final Suggestion* and *Correlated Dishes*: the first is used to return a unique menu, while the second returns all the dishes correlated to the user's constraints.

The separation of the two output variables has, as its main goal, the return of a unique menu to the user composed only by a single dish for each course accompanied by a selection of related dishes, in the case the user does not like the dishes recommended by the main choice. Precisely for this reason, a business knowledge model

Decide Dishes Match Guest Profile
dishList

	inputs	outputs	annotations
U	Guest Profile	Decide Dishes Match Guest Profile	Description
	dietProfileType *"carnivor", "vegetarian", "vegan", "fish-based"*	*dishList* *nullMenu*	
1	"carnivor"	Menu[some i in item.ingredients satisfies i.type = "Meat"]	
2	"vegetarian"	Menu[every i in item.ingredients satisfies not(i.type = "Meat" or i.type = "Fish")]	
3	"vegan"	Menu[every i in item.ingredients satisfies not(i.type = "Meat" or i.type = "Fish" or i.type = "Dairy")]	
4	"fish-based"	Menu[some i in item.ingredients satisfies i.type = "Fish"]	

Figure 2.3: Filter Guest Profile Decision Table

Decide Dishes Match Calories Profile
dishList

	inputs	outputs	annotations
U	Calories Profile	Decide Dishes Match Calories Profile	Description
	caloriesProfileType *"Light/Diet Friedly [200 - 400 kcal]", "Moderate/Balanced [200 - 700 kcal]", "Hearty/Energy rich [200 - 1000 kcal]", "High - Calorie [1000+ kcal]"*	*dishList* *nullMenu*	
1	"Light/Diet Friedly [200 - 400 kcal]"	Menu[sum(item.ingredients.calories) in [200..400]]	
2	"Moderate/Balanced [200 - 700 kcal]"	Menu[sum(item.ingredients.calories) in [200..700]]	
3	"Hearty/Energy rich [200 - 1000 kcal]"	Menu[sum(item.ingredients.calories) in [200..1000]]	
4	"High - Calorie [1000+ kcal]"	Menu[(sum(item.ingredients.calories) in [200..1000]) or (sum(item.ingredients.calories) > 1000)]	

Figure 2.4: Filter Calories Profile Decision Table

called *Personalized-Menu Model*, was implemented in the final solution. Its task is to obtain the first occurrence of a dish for each course and build a final menu, as illustrated in Figure 2.5.

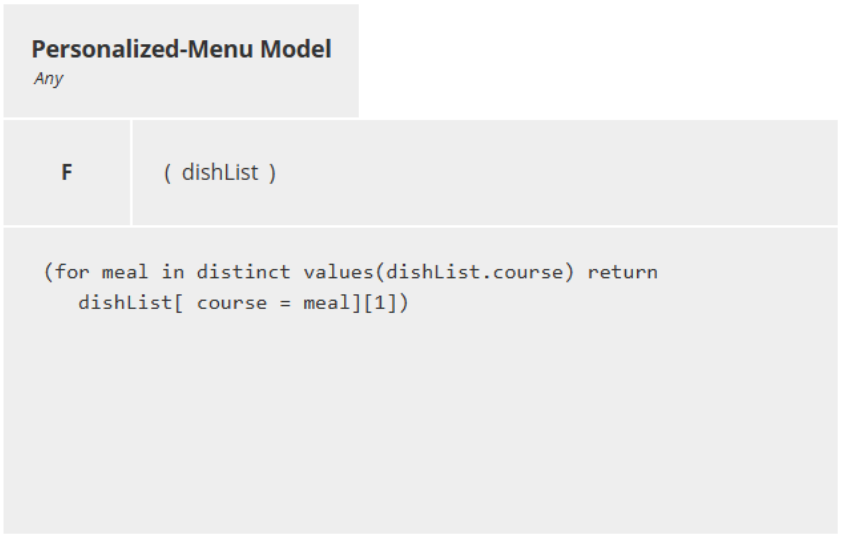

Figure 2.5: Business Knowledge Model adopted

The final decision of *Suggest Personalized Menu* is implemented by two different rules, based on the size of returned filtered list. In particular, if the structure carries at least one element, a final suggestion is returned. Otherwise, a message error is returned for both of the two output variables.

Figure 2.6 presents the final decision table used to provide suggested menu to the user, in particular its structure and how the decision logic was implemented.

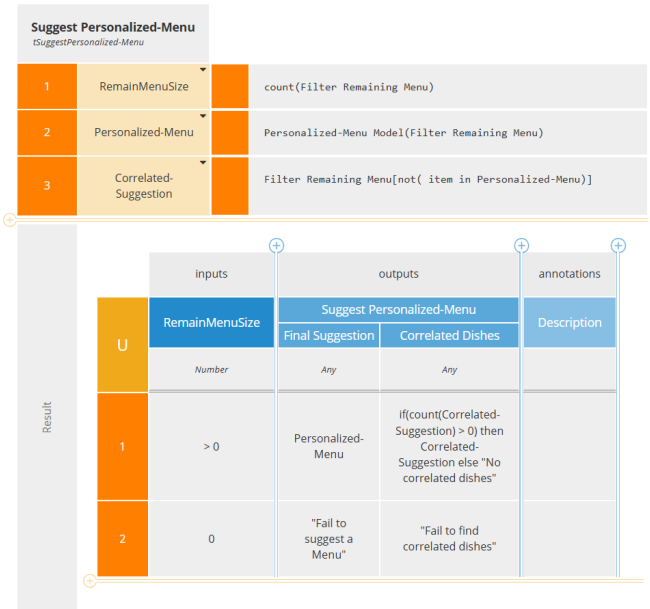

Figure 2.6: Structure of Suggest-Personalized Menu

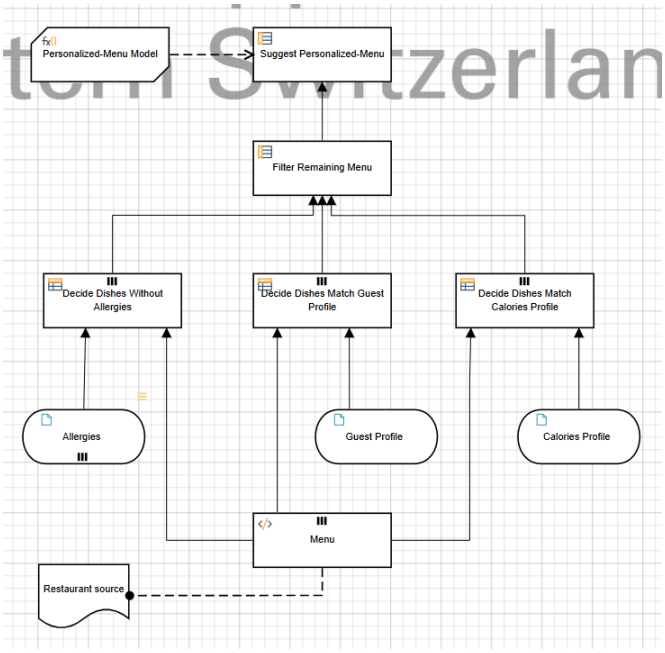

Figure 2.7: Decision Model Diagram

This overall approach is encapsulated in the *Decision Model Diagram* presented in Figure 2.7.

2.2 Prolog

In order to complement the decision-table approach, the personalized-menu engine has been reimplemented in Prolog. The following section illustrates how the core domain knowledge is encoded as Prolog facts and how the recommendation logic is expressed through Prolog rules. All guest preferences, ingredient characteristics and dish compositions are first declared as facts; these are then consumed by the inference rules that compute caloric totals, enforce dietary constraints and filter out allergens.

2.2.1 Facts

The knowledge base opens by asserting each diner's declared profile together with any known allergies. The fragment shown in Figure 2.8 demonstrates how a simple pair of predicates, `guest_profile/1` and `allergy/1`, capture the essential information about what a guest eats and what they must avoid.

Following this, ingredient categories and meal courses are declared. As Figure 2.9 shows, `type/1` classifies each ingredient into broad groups such as `meat`, `fish` or `vegetable`, while `course/1` identifies whether a dish is intended as an appetizer, a first course, a main or a dessert.

At the heart of the fact base is a detailed listing of every ingredient and its caloric content. In Figure 2.10 the predicate `ingredient/1` enumerates each item—ranging from `tomato` and `shrimp` to `mascarpone` and `octopus`—while Figure 2.11 shows how `calories_ingredient/2` associates each ingredient with an approximate kilocalorie value.

```prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%           Guest Profiles and Allergies               %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Guest profiles (using atoms without hyphens, e.g., fish_based instead of fish-based)
guest_profile(carnivor).
guest_profile(vegetarian).
guest_profile(vegan).
guest_profile(fish_based).
guest_profile(calorie_conscious).

% Recognized allergies
allergy(lactose).
allergy(gluten).
allergy(soy).
allergy(eggs).
allergy(peanuts).
allergy(tree_nuts).
allergy(shellfish).
allergy(mollusks).
```

Figure 2.8: Prolog facts defining guest profiles and recognized allergens

```prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%             Ingredient Types and Courses              %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Ingredient types
type(vegetable).
type(dairy).
type(meat).
type(fish).
type(cereal).

% Course types
course(appetizer).
course(first).
course(main).
course(dessert).
```

Figure 2.9: Prolog facts for ingredient types and course categories

```prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                Ingredients and Properties            %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Available ingredients
ingredient(bread).
ingredient(tomato).
ingredient(garlic).
ingredient(basil).
ingredient(rice).
ingredient(shrimp).
ingredient(squid).
ingredient(mussels).
```

Figure 2.10: Prolog listing of ingredients

```
121 % Calorie counts for ingredients (approximate values)
122 calories_ingredient(bread, 250).
123 calories_ingredient(tomato, 18).
124 calories_ingredient(garlic, 149).
125 calories_ingredient(basil, 22).
126 calories_ingredient(rice, 130).
127 calories_ingredient(shrimp, 85).
128 calories_ingredient(squid, 92).
129 calories_ingredient(mussels, 86).
130 calories_ingredient(parsley, 36).
131 calories_ingredient(veal, 110).
132 calories_ingredient(ham, 150).
```

Figure 2.11: Prolog facts for caloric values of ingredients

To link individual ingredients to their categories, the predicate `type_ingredient/2` records, for example, that `shrimp` is of type `fish` and that `eggs` belong to `dairy` (see Figure 2.12). This association is critical both for filtering by diet and for grouping ingredients by allergen.

```
82 % Association between an ingredient and its type
83 type_ingredient(bread, cereal).
84 type_ingredient(tomato, vegetable).
85 type_ingredient(garlic, vegetable).
86 type_ingredient(basil, vegetable).
87 type_ingredient(rice, cereal).
88 type_ingredient(shrimp, fish).
89 type_ingredient(squid, fish).
90 type_ingredient(mussels, fish).
91 type_ingredient(parsley, vegetable).
92 type_ingredient(veal, meat).
93 type_ingredient(ham, meat).
```

Figure 2.12: Prolog associations between ingredients and their types

With all ingredients defined, the set of dishes appears next. Figure 2.13 enumerates each menu item via `dish/1`. The relationship between dishes and ingredients is made explicit by `contains_ingredient/2` facts, illustrated in Figure 2.14, which form the basis for computing nutritional totals. Finally, the predicates `dish_course/2` and `contains_allergy/2`, shown in Figures 2.15 and 2.16, respectively bind each dish to its course type and to the allergens it may contain.

2.2.2 Rules

Having established the fact base, the recommendation engine is implemented via a small set of recursive and composite rules. The predicate `dish_calories/2`, depicted in Figure 2.17, gathers every caloric entry for the ingredients of a given dish and sums them to yield the dish's total energy content. This computed value drives subsequent diet-based checks.

The central logic for generating recommendations resides in the predicate `recommended_dish/3`,

```
164 % Definition of available dishes
165 dish(tomato_bruschetta).
166 dish(seafood_risotto).
167 dish(saltimbocca_alla_romana).
168 dish(tiramisu).
169 dish(caprese_salad).
170 dish(pumpkin_risotto).
171 dish(grilled_salmon).
172 dish(panna_cotta).
173 dish(octopus_salad).
174 dish(lasagna).
175 dish(milanese_cutlet).
```

Figure 2.13: Prolog facts enumerating all available dishes

```
177 % Association of ingredients for each dish
178 contains_ingredient(tomato_bruschetta, bread).
179 contains_ingredient(tomato_bruschetta, tomato).
180 contains_ingredient(tomato_bruschetta, garlic).
181 contains_ingredient(tomato_bruschetta, basil).
182
183 contains_ingredient(seafood_risotto, rice).
184 contains_ingredient(seafood_risotto, shrimp).
185 contains_ingredient(seafood_risotto, squid).
186 contains_ingredient(seafood_risotto, mussels).
187 contains_ingredient(seafood_risotto, garlic).
188 contains_ingredient(seafood_risotto, parsley).
```

Figure 2.14: Prolog facts describing which ingredients each dish contains

```
237 % Association of dishes with course types
238 dish_course(tomato_bruschetta, appetizer).
239 dish_course(seafood_risotto, first).
240 dish_course(saltimbocca_alla_romana, main).
241 dish_course(tiramisu, dessert).
242 dish_course(caprese_salad, appetizer).
243 dish_course(pumpkin_risotto, first).
244 dish_course(grilled_salmon, main).
245 dish_course(panna_cotta, dessert).
246 dish_course(octopus_salad, appetizer).
247 dish_course(lasagna, first).
248 dish_course(milanese_cutlet, main).
```

Figure 2.15: Prolog mapping of dishes to course categories

```
250 % Association of dishes with possible allergen concerns
251 contains_allergy(tomato_bruschetta, gluten).
252
253 contains_allergy(seafood_risotto, shellfish).
254 contains_allergy(seafood_risotto, mollusks).
255
256 contains_allergy(saltimbocca_alla_romana, lactose).
257
258 contains_allergy(tiramisu, lactose).
259 contains_allergy(tiramisu, eggs).
260 contains_allergy(tiramisu, gluten).
261
262 contains_allergy(caprese_salad, lactose).
263
264 contains_allergy(pumpkin_risotto, lactose).
265
266 contains_allergy(panna_cotta, lactose).
267
268 contains_allergy(octopus_salad, mollusks).
269
270 contains_allergy(lasagna, gluten).
271 contains_allergy(lasagna, lactose).
272
273 contains_allergy(milanese_cutlet, gluten).
274 contains_allergy(milanese_cutlet, eggs).
```

Figure 2.16: Prolog annotation of dish-specific allergens

```prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%              Calculation of Dish Calories        %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Calculate the total calories of a dish by summing the calories of its ingredients.
dish_calories(Dish, TotalCalories) :-
    findall(Cal, (contains_ingredient(Dish, Ingredient), calories_ingredient(Ingredient, Cal)), CaloriesList),
    sum_list(CaloriesList, TotalCalories).
```

Figure 2.17: Prolog rule calculating total calories per dish

illustrated in Figure 2.18. This predicate succeeds when a candidate dish satisfies every declared profile constraint, avoids all specified allergens, and naturally falls within any caloric boundaries implied by the profiles. Internally, it invokes the helper predicates `check_profiles/2` and `check_allergies/2`.

```prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%          Rules for Dish Recommendation          %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Main predicate: recommended_dish(Profiles, Allergies, Dish)
% Profiles is a list of profiles that must all be satisfied.
% Allergies is a list of allergens to avoid.
recommended_dish(Profiles, Allergies, Dish) :-
    dish(Dish),
    check_profiles(Profiles, Dish),
    check_allergies(Allergies, Dish).
```

Figure 2.18: Prolog main predicate driving the recommendation system

Profile verification is performed by the composite predicate `check_profiles/2`, which recursively ensures that each profile in the user's list is satisfied by the dish. The individual profile checks, examples of which appear in Figures 2.19 and 2.20, enforce rules such as "no meat for vegetarians," "no dairy or meat for vegans," or a minimum presence of fish for a fish-based diet.

Allergy avoidance is handled by `check_allergies/2`, which succeeds only if none of the undesired allergens is associated with the dish. Through the harmonious interplay of these rules and the foundational facts, the Prolog engine is capable of producing a personalized menu that respects every guest's nutritional goals and health restrictions.

Finally, we created some predefined guest profiles for showing suggested menu during queries thanks to the predicate `suggest_menu/2` as in Figure 2.21.

2.3 Knowledge graph/Ontology

The ontology delivers the same functional coverage requested for the overall project while switching the representation paradigm from rule-centric artefacts (decision tables and `Prolog` facts) to a formally axiomatized knowledge graph grounded in OWL 2 DL. Its conceptualisation has been driven by the textual requirements listed in the project specification, which state that every dish must be described in terms of its ingredients, calories, course and suitability for guest profiles such as vegetarians, vegans, carnivores and allergy sufferers :contentReference[oaicite:0]index=0. A top–down modelling strategy was adopted: competence questions were first rewritten as SPARQL queries and SHACL validation goals; from these, a light upper ontology was derived and later

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%              Composite Profile Checks                        %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% check_profiles(Profiles, Dish) succeeds if Dish satisfies every profile in the Profiles list.
check_profiles([], _).
check_profiles([Profile|Rest], Dish) :-
    check_profile(Profile, Dish),
    check_profiles(Rest, Dish).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%              Profile-based Checks                            %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% For a vegetarian: no ingredient of type meat must be present.
check_profile(vegetarian, Dish) :-
    \+ ( contains_ingredient(Dish, Ingredient),
         type_ingredient(Ingredient, meat)
       ).

% For a vegan: no ingredient of type meat or dairy must be present.
check_profile(vegan, Dish) :-
    \+ ( contains_ingredient(Dish, Ingredient),
         ( type_ingredient(Ingredient, meat) ; type_ingredient(Ingredient, dairy) )
       ).

% For a carnivor: at least one ingredient of type meat should be present.
check_profile(carnivor, Dish) :-
    contains_ingredient(Dish, Ingredient),
    type_ingredient(Ingredient, meat).

% For a fish_based guest: at least one ingredient of type fish should be present.
check_profile(fish_based, Dish) :-
    contains_ingredient(Dish, Ingredient),
    type_ingredient(Ingredient, fish).
```

Figure 2.19: Prolog helper rules for the first set of profile checks

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%          Calorie-based Diet Category Checks                  %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% For a light (diet friendly) profile: the dish must be between 200 and 400 kcal.
check_profile(light, Dish) :-
    dish_calories(Dish, Total),
    Total >= 200,
    Total =< 400.

% For a moderate (balanced) profile: the dish must be between 401 and 700 kcal.
check_profile(moderate, Dish) :-
    dish_calories(Dish, Total),
    Total > 400,
    Total =< 700.

% For a hearty (energy rich) profile: the dish must be between 701 and 1000 kcal.
check_profile(hearty, Dish) :-
    dish_calories(Dish, Total),
    Total > 700,
    Total =< 1000.

% For a high calorie profile: the dish must be above 1000 kcal.
check_profile(high, Dish) :-
    dish_calories(Dish, Total),
    Total > 1000.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%            Allergy-based Checks                              %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% check_allergies(Allergies, Dish) succeeds if none of the allergens in Allergies is present in the Dish.
check_allergies([], _).
check_allergies([Alg | Rest], Dish) :-
    \+ contains_allergy(Dish, Alg),
    check_allergies(Rest, Dish).
```

Figure 2.20: Prolog helper rules for the second set of profile checks

```prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%            Guests-Profile Registered                   %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% suggest_menu(NameGuest, Result) return a menu suggestion based on the user's preferences. Please, consider that
% all the following queries rappresent a pre-built profiles with a set of fixed preferences. If you would obtain a
% custom suggestion it's necessary to execute recommended_dish() query.

suggest_menu("Samuele", Suggest) :- recommended_dish(_, light, _, Suggest).
suggest_menu("Matteo", Suggest) :- recommended_dish(vegetarian, hearty, [eggs, gluten], Suggest).
suggest_menu("Francesco", Suggest) :- recommended_dish(carnivor, high, _, Suggest).
suggest_menu("Enrico", Suggest) :- recommended_dish(vegan, moderate, [lactose], Suggest).
```

Figure 2.21: Suggest Menu Prolog profiles predefined

specialised with the concrete Italian menu supplied in Tasks 1 2.

The resulting TBox is centred on the class `Meal`, with disjoint subclasses `Starter`, `FirstCourse`, `MainCourse` and `Dessert`. The class `Ingredient` is linked to `Meal` through the object property `hasIngredient` (inverse `isIngredientOf`). `Ingredient` is in turn classified by `IngredientType`—an enumeration that mirrors the `type/2` predicate of the Prolog solution—via the functional property `hasType`. Calories are modelled with the datatype property `hasCalories` (domain `Ingredient`, range `xsd:integer`). Intolerances are captured by the class `Allergen`; each `Ingredient` that may trigger an intolerance is connected to the corresponding `Allergen` instance through `mayCause`. Guest preferences are reified by the class `Profile` (vegetarian, vegan, carnivore, fish-based, calorie categories) and related to meals by `suitableFor`. All structural constraints—disjointness, functionality, cardinalities and domain/range axioms—are stated declaratively, allowing an OWL reasoner to check consistency automatically.

2.4 Knowledge graph/Ontology

The definitive ontology for Task 3 is an OWL 2 DL artefact. All entities declared in the file follow that namespace, while the usual `rdf`, `rdfs`, `owl` and `xsd` prefixes are retained for meta-modelling. The file is encoded in Turtle and was authored directly in Protégé, then refined with the OWL API during automated regression tests. The model mirrors the domain already captured by the DMN decision tables and the Prolog facts of as in Sections 2.1 and 2.2, yet it enriches it with formal semantics, reasoning support and an explicit rule layer.

2.4.1 Protégé

In Protégé the first step consisted in declaring the core classes `Dish`, `Ingredient`, `Course`, `Allergy`, `CaloriesDesired`, `FoodProfile` and `Guest`. Nutritional categories (`Meat`, `Fish`, `Dairy`, `Cereal`, `Vegetable`) are modelled as disjoint subclasses of `Ingredient`, an axiom made explicit by an `owl:AllDisjointClasses` construct so that the reasoner will catch mis-classifications. Composite dish types such as `MeatDish`, `FishDish`, `VegetarianDish` and `VeganDish` are defined through qualified cardinality restrictions on the object property `containsIngredient`; for example, `MeatDish` is declared equivalent to "some `containsIngredient Meat`". Object-property modelling follows a systematic inverse pattern: `containsIngredient` is paired with `isContainedBy`, `contains_allergy` with `allergyIsContained`, `belongs_to_the` with `course_includes`, `hasPersonalizedMenuDish` with `inPersonalizedMenu`
Domains and ranges are asserted for every property so that OWL validation can reveal

illegal statements at authoring time.

Quantitative information is attached through datatype properties. Each `Ingredient` carries a mandatory `hasCalories` assertion, while `Dish` instances expose a precomputed `hasTotalCalories` value. Desired calorie bands requested in the specification (*Light*, *Moderate*, *Hearty*, *High*) are represented as individuals of `CaloriesDesired` equipped with `hasMinCalories` and `hasMaxCalories` literals; the open-world assumption is therefore tamed by closing the numeric interval explicitly. The modelling session concluded by instantiating the full menu. Each of the eleven dishes, their forty ingredients, the four courses and the five guest profiles were entered as named individuals; cross-links among them comply with the domains and ranges previously fixed and with the inverses automatically maintained by HermiT. After classification the reasoner placed, for instance, `Grilled_Salmon` under both `Dish` and the inferred subclass `FishDish`, proving that the axioms were coherent.

2.4.2 SWRL

The ontology relies on a compact yet expressive rule layer written in SWRL and activated through the SWRLTab. The annotation property `swrla:isRuleEnabled` is used to toggle individual rules for debugging; all rules shipped with the release are flagged as `true`. One group of rules propagates inverse relations that would otherwise require explicit duplication. The rule labelled *isContained* states that if a dish `x` `containsIngredient` `y`, then `y` `isContainedBy` `x`. A symmetrical pattern establishes `course_includes` from `belongs_to_the`, and `allergyIsContained` from `contains_allergy`. These rules merely materialise data that a SPARQL client can exploit without relying on reasoning at query time.

A second group assigns dishes to qualitative menu profiles. Three single-premise rules map any instance of `MeatDish`, `FishDish` or `VeganDish` to the appropriate `FoodProfile` via the object property `suggested_for`. Two further rules with identical heads are provided so that both variable-bound and constant-bound formulations can be toggled while measuring performance. A third group calculates diet suitability based on total calories. The rule labelled *Deduce Calories Profile* takes a dish `d` together with its `hasTotalCalories` value, compares that literal against the `hasMinCalories` and `hasMaxCalories` of a calories band `p` through built-in arithmetic atoms, and if the interval comparison succeeds it asserts `hasDesiredCaloriesDishes(p, d)`; the companion rule then infers `inDesiredCaloriesProfile(d, p)` to allow OWL class expressions to refer to such dishes directly. All rules have been sanity-checked under the Hybrid reasoner and produce exactly the triples expected from the manual sample calculations in Tasks in Sections 2.1 and 2.2.

The ontology is therefore ready for deployment in any triple store that supports SWRL materialisation, SHACL validation and SPARQL 1.1 querying. It exposes richer semantics than the previous artefacts while remaining completely aligned with them, thereby ensuring that downstream BPMN processes can retrieve personalised menus by a single parameterised SPARQL query, free from procedural glue code.

2.4.3 SPARQL Queries

The ontology is consumed through a small set of parameterised SPARQL 1.1 queries that transform the inferred triples into the artefacts needed by the personalised–menu application. Each query is designed for direct execution in a Fuseki endpoint; names

and IRIs are written in the same namespace used by the OWL file so that no additional prefixes are required. Where run-time parameters occur, they are supplied via `VALUES` clauses rather than string concatenation, a strategy that keeps the query text itself static and cache-friendly while still allowing simple injection from the BPMN engine. All filters that might discard a solution are expressed through `FILTER NOT EXISTS` patterns, thereby preserving monotonicity under entailment. The queries shipped with Task 3 are reported verbatim below.:contentReference[oaicite:0]index=0

Query 1 – Retrieve vegetarian, moderate-calorie dishes that avoid a given allergy. The first query returns the triple `(?dish, ?course, ?calories)` for every dish classified as `Vegetarian`, belonging to the calorie band `Moderate` and not containing the allergy passed in the `?excludedAllergy` variable. Two `BIND` expressions translate the matching of the food profile and the calorie profile into Boolean flags; their product (`?suggestion`) is then compared against 0, ensuring that a row survives the filter only when both aspects are satisfied.

```
PREFIX : <http://www.semanticweb.org/kebi.task3.pirani-machella#>

SELECT ?dish ?course ?calories
WHERE {
  VALUES ?desiredFoodProfile   { :Vegetarian }
  VALUES ?desiredCalorieProfile { :Moderate }
  VALUES ?excludedAllergy      { :Shellfish }

  ?dish a :Dish ;
        :hasTotalCalories ?calories ;
        :belongs_to_the ?course ;
        :suggested_for ?foodprofile ;
        :inDesiredCaloriesProfile ?caloriesprofile .

  BIND(IF(?foodprofile     = ?desiredFoodProfile,   1, 0) AS ?v1)
  BIND(IF(?caloriesprofile = ?desiredCalorieProfile, 1, 0) AS ?v2)
  BIND((?v1 * ?v2) AS ?suggestion)

  FILTER(?suggestion > 0)
  FILTER NOT EXISTS { ?dish :contains_allergy ?excludedAllergy }
}
```

Query 2 – Construct a personalised menu for every guest present in the graph. The second query is formulated in the `CONSTRUCT` form so that its result can be inserted back into the triple store as materialised links between guests and suitable dishes. It iterates simultaneously over all `:Guest` and `:Dish` individuals, filters away dishes that violate declared allergies, and checks whether both the food profile and the calorie profile required by the guest coincide with those of the dish. Missing requirements are tolerated thanks to `OPTIONAL` and `FILTER` conditions that treat unbound variables as "no constraint".

```
PREFIX : <http://www.semanticweb.org/kebi.task3.pirani-machella#>
```

```
CONSTRUCT {
  ?guest :hasPersonalizedMenuDish ?dish .
}
WHERE {
  ?guest a :Guest .
  ?dish  a :Dish .

  OPTIONAL { ?guest :guestHasFoodProfile     ?fp  }
  OPTIONAL { ?dish  :suggested_for          ?fp_d }

  OPTIONAL { ?guest :guestHasCaloriesDesired ?cp  }
  OPTIONAL { ?dish  :inDesiredCaloriesProfile ?cp_d }

  FILTER NOT EXISTS { ?guest :guestSufferFrom ?a .
                      ?dish  :contains_allergy ?a . }

  FILTER (!BOUND(?fp) || ?fp = ?fp_d)
  FILTER (!BOUND(?cp) || ?cp = ?cp_d)
}
```

Query 3 – Construct a personalised menu for a specific guest. When the calling application already knows which guest is active, the previous query can be specialised by binding the `?guest` variable to a constant—`:Samuele` in the example—through a `BIND` expression. The remainder of the query is identical, guaranteeing consistent semantics across both use cases. This version is particularly convenient for embedding directly into a BPMN task where the guest identifier is available as a process variable.

```
PREFIX : <http://www.semanticweb.org/kebi.task3.pirani-machella#>

CONSTRUCT {
  ?guest :hasPersonalizedMenuDish ?dish .
}
WHERE {
  BIND(:Samuele AS ?guest)
  ?dish a :Dish .

  OPTIONAL { ?guest :guestHasFoodProfile     ?fp  }
  OPTIONAL { ?dish  :suggested_for          ?fp_d }

  OPTIONAL { ?guest :guestHasCaloriesDesired ?cp  }
  OPTIONAL { ?dish  :inDesiredCaloriesProfile ?cp_d }

  FILTER NOT EXISTS { ?guest :guestSufferFrom ?a .
                      ?dish  :contains_allergy ?a . }

  FILTER (!BOUND(?fp) || ?fp = ?fp_d)
  FILTER (!BOUND(?cp) || ?cp = ?cp_d)
}
```

The three queries together complete the knowledge-based stack: SWRL materialises the implicit `suggested_for` and `inDesiredCaloriesProfile` links, while SPARQL uses that information to assemble a guest-specific menu, ready to be streamed into the mobile front-end or injected into the BPMN execution context without any additional transformation.

2.4.4 SHACL Shapes

To guarantee that every RDF graph uploaded to the triplestore respects the structural and numerical constraints that underpin menu reasoning, a dedicated SHACL shape graph has been authored in the same namespace as the ontology. The validation layer is designed to be both strict enough to detect modelling mistakes early and permissive enough to support incremental enrichment of the dataset. Each domain class that can be instantiated by end-users—`Dish`, `Ingredient`, `Course`, `FoodProfile`, `CaloriesDesired` and `Guest`—is associated with a `sh:NodeShape`. The shapes use closed-world cardinality checks for mandatory links, datatype restrictions for numeric literals and range guards such as `sh:minInclusive` 0 to rule out negative calorie values before they reach the SWRL layer. Optional relationships are bound by `sh:maxCount` rather than hard `sh:minCount` so that their absence signals "no preference" rather than an error, thereby aligning SHACL validation with the semantics of the SPARQL queries that treat unbound variables as wildcards.

The following Turtle fragment shows the complete shape graph shipped; it can be loaded into any SHACL-aware engine—Fuseki, GraphDB or PySHACL—for batch validation or plugged into real-time data-entry pipelines to provide immediate feedback.

```
@prefix :      <http://www.semanticweb.org/kebi.task3.pirani-machella#> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix sh:    <http://www.w3.org/ns/shacl#> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:   <http://www.w3.org/2002/07/owl#> .

:DishShape a sh:NodeShape ;
  sh:targetClass :Dish ;
  sh:property [
      sh:path :belongs_to_the ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
  ] ;
  sh:property [
      sh:path :containsIngredient ;
      sh:minCount 1 ;
  ] ;
  sh:property [
      sh:path :suggested_for ;
      sh:minCount 1 ;
  ] ;
  sh:property [
      sh:path :inDesiredCaloriesProfile ;
      sh:minCount 1 ;
```

```
      sh:maxCount 1 ;
  ] ;
  sh:property [
      sh:path :hasTotalCalories ;
      sh:datatype xsd:integer ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:minInclusive 0 ;
  ] .

:IngredientShape a sh:NodeShape ;
  sh:targetClass :Ingredient ;
  sh:property [
      sh:path :hasCalories ;
      sh:datatype xsd:integer ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:minInclusive 0 ;
  ] ;
  sh:property [
      sh:path :isContainedBy ;
      sh:minCount 1 ;
  ] .

:CourseShape a sh:NodeShape ;
  sh:targetClass :Course ;
  sh:property [
      sh:path :course_includes ;
      sh:minCount 1 ;
  ] .

:FoodProfileShape a sh:NodeShape ;
  sh:targetClass :FoodProfile ;
  sh:property [
      sh:path :includeDishes ;
      sh:minCount 1 ;
  ] .

:CaloriesDesiredShape a sh:NodeShape ;
  sh:targetClass :CaloriesDesired ;
  sh:property [
      sh:path :hasDesiredCaloriesDishes ;
      sh:minCount 1 ;
  ] ;
  sh:property [
      sh:path :hasMinCalories ;
      sh:datatype xsd:integer ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
```

```
      sh:minInclusive 0 ;
  ] ;
  sh:property [
      sh:path :hasMaxCalories ;
      sh:datatype xsd:integer ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:minInclusive 0 ;
  ] .

:GuestShape a sh:NodeShape ;
  sh:targetClass :Guest ;
  sh:property [
      sh:path :guestHasFoodProfile ;
      sh:maxCount 1 ;
  ] ;
  sh:property [
      sh:path :guestHasCaloriesDesired ;
      sh:maxCount 1 ;
  ] .
```

Together with the OWL axioms and SWRL rules, these SHACL shapes complete the three-layer knowledge representation stack: OWL for closed-world conceptual integrity, SWRL for deductive enrichment and SHACL for runtime data quality assurance. The result is a robust, self-validating knowledge graph that can safely drive personalised menu recommendations in production.

2.5 Summary

This chapter has outlined the progressive refinement of the personalised-menu engine from rule-based artefacts to a fully fledged knowledge graph. Decision Tables were introduced first to partition the restaurant's dish list along three orthogonal dimensions—allergenic safety, dietary suitability and caloric adequacy—so that each constraint could be maintained, tested and evolved independently. The same logic was then re-expressed in Prolog, where explicit facts captured every ingredient, dish, guest preference and allergen, while recursive rules computed calorie totals and enforced the composite filtering conditions required for a valid recommendation.

Building on this foundation, the solution migrated to an OWL 2 DL ontology that preserves functional equivalence with the earlier artefacts yet enriches them with formal semantics. Protégé was used to declare the core classes, disjoint ingredient categories, inverse object properties and quantitative data properties; SWRL rules materialise inverse relations, infer diet profiles and assign dishes to calorie bands without duplicating data; SPARQL 1.1 queries transform the inferred triples into guest-specific menus ready for downstream BPMN tasks; and a dedicated SHACL shape graph enforces structural integrity and numeric bounds at ingestion time. Collectively, these layers provide a coherent and self-validating knowledge-based solution in which declarative constraints, deductive enrichment and query-time retrieval cooperate seamlessly to deliver accurate, explainable and maintainable recommendations.

3. Agile and ontology-based meta-modeling

4. Conclusions

4.1 Matteo Machella's Considerations

4.2 Samuele Pirani's Considerations