Adaptation of TAAABLE to the CCC'2017 Mixology and Salad Challenges, adaptation of the cocktail names

Emmanuelle Gaillard, Jean Lieber, and Emmanuel Nauer

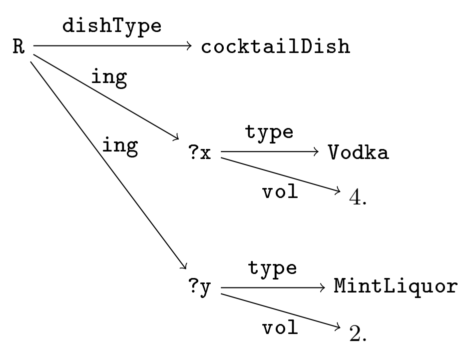
UL, CNRS, Inria, Loria, F-54000 Nancy, firstname.lastname@loria.fr

Abstract. This paper presents the submission of the TAAABLE team to the 2017 Computer Cooking Contest. All challenges except the sandwich challenge are addressed. Online systems have been developed for the *salad* and *mixology* challenges by adapting previous successful CCC TAAABLE systems to the requirements of the 2017 challenges. However, this paper presents two main contributions. The first contribution is a new approach based on adaptation rules for managing the ingredients available for adapting salad recipes, and the second contribution is a work on cocktail name adaptation which is submitted to the *open challenge*. The cocktail name adaptation takes into account the name of the recipe which has to be adapted, knowledge about the name of the recipe and about the substituted and substituting ingredients. The naming process uses the problem-solution dependency to the context of the target problem, exploiting knowledge encoded in RDFS, a semantic Web representation language. This approach in the framework of the representation language RDFS is a general approach and can be applied in other problem solving contexts.
Keywords: case-based reasoning, adaptation, cooking, RDFS.

1 Introduction

This paper presents the participation of the TAAABLE team to the 2017 Computer Cooking Contest. TAAABLE addresses all the challenges except the sandwich challenge. Online systems have been developed for the *salad* and *mixology* challenges, respectively available on `http://tuuurbine.loria.fr/taaableCCC2017/salad.php` and `http://tuuurbine.loria.fr/taaableCCC2017/cocktail.php`. For these two challenges, previous successful CCC TAAABLE systems have been adapted to fulfill the requirements of the 2017 challenges. The management of a limited set of ingredients (called *the fridge* in the following) uses, for the mixology challenge, the process that has been presented at the CCC 2015 contest. Once the basic retrieval process of TAAABLE returns a recipe to adapt, this process based on formal concept analysis searches the more accurate available food which has to be used to substitute for a non available one. A first contribution of this paper concerns the salad challenge, for which we propose a new approach to manage the fridge, using adaptation rules. The second main contribution of this year concerns the *open challenge* for which we present a work about cocktail name adaptation. The cocktail name adaptation takes into account the

Copyright © 2017 for this paper by its authors. Copying permitted for private and academic purpose. In Proceedings of the ICCBR 2017 Workshops. Trondheim, Norway



The "Green Russian" recipe is identified by the resource `R`. Its ingredients are 4 cl of vodka and 2 cl of mint liquor. The preparation is not represented. `ing` relates a recipe to one of its ingredients. `type` (abbreviation of `rdf:type`) is an RDF property relating a class to its instance (for example, the triple ⟨`?x type vodka`⟩ means that `?x` is an instance of `vodka`. The variables are existentially quantified (there exist `?x` and `?y` such that...). The property `vol` relates an ingredient to its volume in centilitres.

Fig. 1. An RDF graph representing the "Green Russian" cocktail recipe.

name of the recipe which has been adapted, knowledge about the name of the recipe and about the substituted and substituting ingredients. The naming process uses the problem-solution dependency to the context of the target problem, exploiting knowledge encoded in RDFS, a semantic Web representation language. This approach in the framework of the representation language RDFS is a general approach and can be applied in other problem solving contexts.

The paper is organized as follows: Section 2 introduces the core of the TAAABLE system. Sections 3, 4 and 5 present the TAAABLE systems which adress respectively the mixology, the salad and the open challenges.

2 The TAAABLE system

The challenges, proposed by the CCC since its first edition consists in proposing, according to a set of initial recipes, one or more recipes matching a user query composed of a set of wanted ingredients and a set of unwanted ingredients. Since 2015, the TAAABLE systems are built using TUUURBINE [1] (`http://tuuurbine.loria.fr`), a generic case-based reasoning (CBR) system over RDFS which allows reasoning with knowledge stored in a RDF store, as the one provided by the contest. TUUURBINE implements a generic CBR mechanism in which adaptation consists in retrieving similar cases and in replacing some features of these cases in order to adapt them as a solution to a query.

2.1 RDFS

RDF (*Resource Description Framework*) represents data as triples of resources ⟨*subject predicate object*⟩, where the resource *predicate* is a property. A resource is either a constant or a variable (generally called identified resource and blank node, respectively). By naming convention, variables start with the symbol `?` whereas constants do not. Fig. 1 illustrates a recipe represented using RDF triples.

RDFS gives some semantics—and thus, inference possibilities—to RDF by the mean of inference rules associated to some resources. Only a few rules are used in this paper:

$$\frac{\langle \texttt{a type C} \rangle \quad \langle \texttt{C subc D} \rangle}{\langle \texttt{a type D} \rangle} r_1 \quad \frac{\langle \texttt{a p b} \rangle \quad \langle \texttt{p subp q} \rangle}{\langle \texttt{a q b} \rangle} r_2$$

$$\frac{\langle \texttt{A subc B} \rangle \quad \langle \texttt{B subc C} \rangle}{\langle \texttt{A subc C} \rangle} r_3 \quad \frac{\langle p \texttt{ subc } q \rangle \quad \langle q \texttt{ subc } r \rangle}{\langle p \texttt{ subc } r \rangle} r_4$$

`type`, `subc` and `subp` are abbreviations for `rdf:type`, `rdfs:subClassOf` and `rdfs:subPropertyOf`. `type` is the membership relation between an instance and a class. `subc` (resp., `subp`) is the relation between a class and a superclass (resp., a property and a superproperty). r_1 means that if `a` is an instance of a class it is also an instance of its superclasses. r_2 means that if `a` and `b` are related by a property, they are also related by any of its superproperties. r_3 and r_4 state that `subc` and `subp` are transitive. For example, the following inference can be drawn:

$$\left\{ \begin{array}{c} \langle \texttt{?x type Vodka} \rangle, \\ \langle \texttt{vodka subc Alcohol} \rangle \end{array} \right\} \vdash \langle \texttt{?x type Alcohol} \rangle$$

RDFS does not include negation, thus only positive facts can be entailed. However, an inference with closed world assumption (CWA) can be drawn, stating that if $\mathcal{B} \not\vdash t$ then t is considered to be false (given the RDFS base $\mathcal{B}$), denoted by $\mathcal{B} \vdash_{\text{cwa}} \neg t$.

SPARQL (*SPARQL Protocol and RDF Query Language*) enables to write queries to RDF or RDFS bases. If a SPARQL engine uses RDFS entailment, this means that the query is done on the RDF base completed by RDFS entailment. For example, the following SPARQL query addressed to a base describing recipes such as the one of figure 1 returns the set of recipes `?r` containing some alcohol, taking into account the domain knowledge, in particular the subclass relations of the food hierarchy presented in Fig. 2. The CWA is assumed: if it cannot be entailed that a recipe contains some alcohol, then it is concluded that it does not.

$$\texttt{Q}_{\texttt{alcohol}} = \text{SELECT } \texttt{?r} \text{ WHERE } \{\texttt{?r ing ?a . ?a type Alcohol}\} \tag{1}$$

Given a SPARQL query `Q` and an RDFS base $\mathcal{B}$, the result of the execution of `Q` on $\mathcal{B}$ is denoted by $\texttt{exec}_{\vdash}(\texttt{Q}, \mathcal{B})$.

2.2 TUUURBINE founding principles

TUUURBINE is a generic CBR system over RDFS .

The **domain knowledge** is represented by an RDFS base `DK` consisting of a set of triples of the form $\langle C \texttt{ subc } D \rangle$ where C and D are classes which belong to a same hierarchy (e.g, the food hierachy). Fig. 2 represents the domain knowledge for the running examples by a hierarchy whose edges $C \xrightarrow{x} D$ represent the triples $\langle C \texttt{ subc } D \rangle$ with x, the retrieval knowledge encoded by a cost function $\texttt{cost}(\langle C \texttt{ subc } D \rangle) = x$. This cost can be understood intuitively as the measure of "the generalization effort" from C to D. How this cost is computed is detailed in [2].

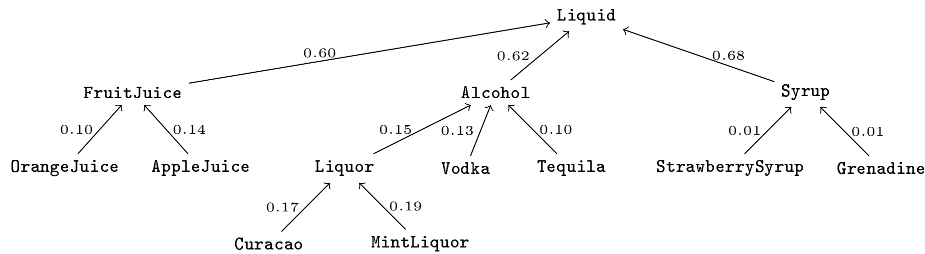


Fig. 2. The hierarchy forming the domain knowledge used in the running example with the generalization costs used as retrieval knowledge.

A **TUUURBINE case** `case` is described by a set of triples of the form $\langle URI_{\texttt{case}}\ \texttt{prop}\ \texttt{val}\rangle$, where $URI_{\texttt{case}}$ is the URI of `case`, `val` is either a resource representing a class of the ontology or a value and `prop` is an RDF property linking `case` to a hierarchy class or to the value.

For simplification, in this paper, we represent a case by a conjunction of expressions only of the form `prop : val`. For example, the "Green Russian" recipe is represented by the following index `R`, which means that "Green Russian" is a cocktail recipe made from vodka and mint liquor (`ing` stands for *ingredient*).

$$\texttt{R} = \texttt{dishType}:\texttt{CocktailDish} \wedge \texttt{ing}:\texttt{Vodka} \wedge \texttt{ing}:\texttt{MintLiquor} \qquad (2)$$

For instance, the first conjunct of this expression means that the triple $\langle URI_R\ \texttt{dishType}\ \texttt{CocktailDish}\rangle$ belongs to the knowledge base.

2.3 TUUURBINE query

A TUUURBINE query is a conjunction of expressions of the form `sign prop:val` where $\texttt{sign} \in \{\epsilon, +, !, -\}$, `val` is a resource representing a class of the ontology and `prop` is an RDF property belonging to the set of properties used to represent cases. For example,

$$\texttt{Q} = +\texttt{dishType}:\texttt{CocktailDish} \wedge \texttt{ing}:\texttt{Vodka} \wedge !\texttt{ing}:\texttt{Grenadine} \qquad (3)$$

is a query to search "a cocktail with vodka but without grenadine".

The signs ϵ (*empty sign*) and $+$ are "positive signs": they prefix features that the requested case must have. $+$ indicates that this feature must also occur in the source case whereas ϵ indicates that the source case may not have this feature, thus the adaptation step has to make it appear in the final case. For example, the term $+$`dishTypeCocktailDish` means that TUUURBINE will only retrieve cases which are cocktail recipes.

The signs ! and $-$ are "negative signs": they prefix features that the requested case must not have. $-$ indicates that this feature must not occur in the source case whereas ! indicates that the source case may have this feature, and, if so, that the adaptation step has to remove it.

2.4 TUUURBINE retrieval process

The retrieval process consists in searching for cases that best match the query. If an exact match exists, the corresponding cases are returned. For the query Q given in (3), the “Green Russian” recipe is retrieved without adaptation. Otherwise, the query is relaxed using a generalization function composed of one-step generalizations, which transforms Q (with a minimal cost) until at least one recipe of the case base matches $\Gamma(Q)$. A one step-generalization is denoted by $\gamma = \texttt{prop} : \texttt{A} \rightsquigarrow \texttt{prop} : \texttt{B}$, where A and B are classes belonging to the same hierarchy with $\texttt{A} \sqsubseteq \texttt{B}$, and prop is a property used in the case definition. This one step-generalization can be applied only if A is prefixed by ϵ or ! in Q. If A is prefixed by !, thus B is necessarily the top class of the hierarchy. For example, the generalization of $!\texttt{ing} : \texttt{Grenadine}$ is $\epsilon\texttt{ing} : \texttt{Food}$, meaning that if grenadine is not wanted, it has to be replaced by some other food or to be removed. Classes of the query prefixed by $+$ and $-$ cannot be generalized.

Each one-step generalization is associated with a cost denoted by $\texttt{cost}(A \rightsquigarrow B)$. The generalization Γ of Q is a composition of one-step generalizations $\gamma_1, \ldots \gamma_n$: $\Gamma = \gamma_n \circ \ldots \circ \gamma_1$, with $\texttt{cost}(\Gamma) = \sum_{i=1}^{n} \texttt{cost}(\gamma_i)$. For example, for:

$$\begin{aligned} \texttt{Q} = &+\texttt{dishType}:\texttt{CocktailDish} \\ &\wedge \texttt{ing}:\texttt{Vodka} \wedge \texttt{ing}:\texttt{Curacao} \wedge !\texttt{ing}:\texttt{Grenadine} \end{aligned} \tag{4}$$

Curacao is relaxed to Liquor according to the domain knowledge of Fig. 2. At this first step of generalization, $\Gamma(\texttt{Q}) = \texttt{dishType}:\texttt{CocktailDish} \wedge \texttt{ing}:\texttt{Vodka} \wedge \texttt{ing}:\texttt{Liquor} \wedge !\texttt{ing}:\texttt{Grenadine}$, which matches the recipe described in (1), indexed by MintLiquor, which is a Liquor.

2.5 TUUURBINE adaptation process

When the initial query does not match existing cases, the cases retrieved after generalization have to be adapted. The adaptation consists of a specialization of the generalized query produced by the retrieval step. According to $\Gamma(\texttt{Q})$, to R, and to DK, the ingredient MintLiquor is replaced by the ingredient Curacao in R because Liquor of $\Gamma(\texttt{Q})$ subsumes both MintLiquor and Curacao. So, the adaptation consists in replacing curacao by mint liquor.

TUUURBINE implements also an adaptation based on rules where a rule states that in a given context $\mathcal{C}$, some ingredients $\mathcal{F}$ can be replaced by other ingredients $\mathcal{B}$. $\mathcal{C}$, $\mathcal{F}$ and $\mathcal{B}$ are the contexts, the “from part” (premise) and the “by part” (conclusion) of the adaptation rule [3]. For example, the piece of knowledge stating that, in cocktail recipes, orange juice and strawberry syrup can be replaced with pineapple juice and grenadine, can be represented by an adaptation rule with $\mathcal{C} = \texttt{CocktailDish}$, $\mathcal{F} = \texttt{OrangeJuice} \wedge \texttt{StrawberrySyrup}$ and $\mathcal{B} = \texttt{PineappleJuice} \wedge \texttt{Grenadine}$. Such an adaptation rule can be encoded by a substitution $\sigma = \mathcal{C} \wedge \mathcal{F} \rightsquigarrow \mathcal{C} \wedge \mathcal{B}$. In the example: $\texttt{CocktailDish} \wedge \texttt{OrangeJuice} \wedge \texttt{StrawberrySyrup} \rightsquigarrow \texttt{CocktailDish} \wedge \texttt{PineappleJuice} \wedge \texttt{Grenadine}$. This rule-based adaptation is directly integrated in the retrieval process by searching cases indexed by the substituted ingredients for a query about the replacing ingredients, for example by searching recipes containing OrangeJuice and StrawberrySyrup for a query about PineappleJuice and Grenadine.

2.6 TAAABLE as a TUUURBINE instantiation

The TAAABLE knowledge base is WIKITAAABLE (http://wikitaaable.loria.fr/); WIKITAAABLE is composed of the four classical knowledge containers: (1) the domain knowledge contains an ontology of the cooking domain which includes several hierarchies (about food, dish types, etc.), (2) the case base contains recipes described by their titles, the dish type they produce, the ingredients that are required, the preparation steps, etc., (3) the adaptation knowledge takes the form of adaptation rules as introduced previously, and (4) the retrieval knowledge, which is stored as cost values on subclass-of relations and adaptation rules.

In WIKITAAABLE, all the knowledge (cases, domain knowledge, costs, adaptation rules) is stored into a triple store. So, plugging TUUURBINE over the WIKITAAABLE requires only configuring TUUURBINE by giving the case base root URI, the ontology root URI and the set of properties on which reasoning may be applied.

3 Mixology challenge

The mixology challenge consists in retrieving a cocktail that matches a user query according to a set of available foods given by the CCC organizers. For this challenge, the successful TAAABLE system which won the jury and public prizes in 2015 has been adapted to take into account the new list of available foods (vodka, gin, rum, tequila, sake, champagne, tomato juice, apple juice, sparkling water, grenadine syrup, lemon juice, lime, mint, ice cube, brown sugar, salt, pepper). The principles remain the same as the ones used in the 2015 system. TUUURBINE is used to perform the retrieval step which takes into account the available foods (section 3.1). A specific adaptation step based on formal concepts is used when some ingredients of the source recipe are not available, to search the best way to replace them, or in some cases, to remove them (see Section 3.2).

3.1 Managing a fridge with TUUURBINE

TUUURBINE is able to manage a fridge directly through a query modification using the ϵ and ! prefixes. Indeed, if answers must only contain the available foods, the initial user query can be modified by adding the minimal set of classes of the food hierarchy that subsume the set of foods which are not available, each class being negatively prefixed by !. For example, let us assume that `OrangeJuice` and `AppleJuice` are the available fruit juices, that `Vodka` and `Tequila` are the only available alcohols, that `Grenadine` is the only available syrup, and that the user wants a cocktail recipe with `Vodka` but without `Grenadine`. The initial user query will be $\texttt{Q} = +\texttt{dishType}:\texttt{CocktailDish} \wedge \epsilon\texttt{ing}:\texttt{Vodka} \wedge !\texttt{ing}:\texttt{Grenadine}$. According to Fig. 2, `Liquor` and `StrawberrySyrup` will be added to this initial query with a ! for expressing that the result cannot contain one of these non available classes of food. The extended query `EQ` submitted to TUUURBINE will be:

$$\texttt{EQ} = \texttt{Q} \wedge !\texttt{ing}:\texttt{Liquor} \wedge !\texttt{ing}:\texttt{StrawberrySyrup}$$

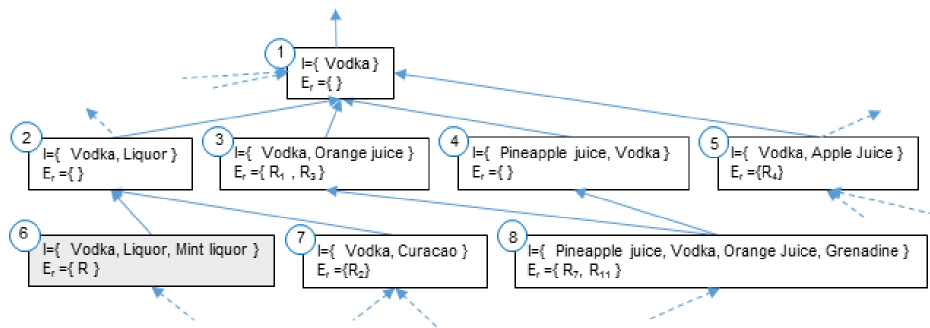


Fig. 3. Part of the concept lattice built from recipes using `Vodka`.

For this example, TUUURBINE retrieves the "Green Russian" recipe with the adaptation "replace `MintLiquor` with `Food`". In order to replace `MintLiquor` by something more specific than `Food`, a FCA approach which exploits ingredient combination in cocktail recipes is used.

3.2 Using FCA to search the best ingredient combination

FCA is a classification method allowing object grouping according to the properties they share [4] and so, is able to find regularities in a set of objects. When the objects are cocktail recipes and the properties are the ingredients the cocktail recipes use, FCA computes ingredient combination. FCA produces *formal concepts* as output. A formal concept is a pair (I, E) where I is a set of properties, E is a set of objects, respectively called the *intent* and the *extent* of the formal concept, such that (1) I is the set of all properties shared by the objects of E and (2) E is the set of all objects sharing properties in I. The formal concepts can be ordered by extent inclusion, also called *specialisation* between concepts, into what is called a *concept lattice*. Fig. 3 illustrates a part of the lattice resulting from recipes with `Vodka` (the only required ingredient in `Q`). On this figure, the extents E are given through a reduced form (noted E_r): the objects appear in the most specific concepts, the complete extent can be computed by the union of objects belonging to the subconcepts. For example, the concept #3 is related to cocktail recipes using vodka and orange juice. These recipes are R_1 and R_3 which do not contain other ingredient, and the recipes of the extents of concept #3 more specific concepts (e.g. concept #8) which contain additional ingredients. This lattice can be used to adapt the "Green Russian" recipe `R`, returned for query `Q`, with the substitution of `MintLiquor` with *another food* because `MintLiquor` is not in the set of available foods.

To search a replacing ingredient in a given recipe or in a recipe according to pieces of food that will be kept, the idea is to exploit the lattice which captures concept similarities and organization. Adapting a cocktail is based on the closeness between concepts. For example, when a replacing ingredient is searched for `MintLiquor` in `R` (concept #6), some similar concepts (i.e. sharing a same super-concept) can be used. In the lattice given in example, concept #6 can be generalized into concept number #2, which extent contains cocktails with vodka and liquor. The cocktail in the extent of concept #7

is similar to the one of concept #2, because they share the Vodka and the Liquor properties. When removing MintLiquor from the "Green Russian" recipe, a possible ingredient for substitution, given by the lattice, could be Curacao. However, as Curacao is not an available food, concept #7 cannot be used to complete the substitution.

Let C_R be the formal concept such that $E_r(C_R) = \{R\}$. A formal concept C close to C_R is searched according to the following procedure. C is such that its intent $I(C)$ does not contain the removed ingredient (MintLiquor in the example) and maximizes $|E_r(C)|$. First, C is searched in the ascendants of C_R, then in the descendants of the ascendants satisfying the available food constraints. The ingredient to be substituted is replaced by $I(C) \setminus I(C_R)$.

Applying this procedure, the most similar ingredient combinations which includes Vodka that can be used to replace MintLiquor are given by concepts #3, #4 and #5 and their descendants. However, concept #4 and its descendants cannot be used to produce a substitution because its intent contains PineappleJuice which is not an available food. Concept #5 intent contains AppleJuice, an available food, but concept #3 is closer to concept #6 than concept #5 is, according to the selection procedure based on the maximal number of objects of E_r. The cocktail system will suggest replacing the mint liquor with orange juice.

To implement our approach, data about ingredient combinations in cocktail recipes has been collected. For this, we queried *Yummly* (http://www.yummly.com/) with query composed of one ingredient (one available food from the CCC 2017 new food list). More details about the FCA based approach can be found in [5].

4 Salad challenge: a new approach for managing the fridge and for adapting quantities

The adaptation challenge requires managing a limited set of available food (like in the cocktail challenge) and adapting the ingredient quantities.

4.1 Salad ingredient adaptation with a fridge

The approach for managing a limited set of ingredients is rather different in the salad challenge context than in the cocktail one. Indeed, there are many important differences between the knowledge involved to solve a recipe adaptation for the mixology challenge and the knowledge involved to solve a recipe adaptation for the salad challenge. First, there are less salad recipes than cocktails recipes: 70 against 108. Second, the salad recipes use 266 different ingredients of whom 245 are not available in the fridge. For the cocktail recipes, only 139 ingredients (among the 156 different ingredients) are not in the fridge. Third, the minimal, maximal and average of ingredients per recipe is 4, 18 and 10 for the salad recipes and only 2, 10 and 5 for the cocktail recipes. The second and third points directly impact the number of ingredient substitutions required to adapt a recipe to fit the fridge constraint. Adapting a salad recipe requires at least 3 substitutions and in average 8 substitutions (the maximal number of substitutions is 17). For the cocktail recipes, to take into account the fridge, the maximal number of substitutions is only 7, 10 recipes require only 1 ingredient substitution and, in average,

the number of substitutions is 3. Fifth, the ingredients involved in the salad recipes are more distributed over the food hierarchy: 13 top categories of the food hierarchy are concerned (Vegetable, Fruit, Meat, Seafood, Legume, Diary, Liquid, Oil, ...), whereas only 3 top categories are concerned for cocktail recipes (Liquid, Fruit, and Flavoring). All these facts drastically increase the adaptation effort for the salad challenge comparing to the cocktail challenge: more ingredients to substitute in order to adapt a recipe, more unavailable foods implied and a larger distribution of this foods in the food hierarchy. So, it is not appropriate to generalize an unavailable food to Food because too many food classes will be generalized into Food: 8 in average. With the approach presented for adapting cocktail recipes, it requires searching for each of these 8 ingredients which ingredients of the fridge can be used as substituting ingredients. That is why we propose a new approach using adaptation rules, in order to better control the adaptation. The idea is to define, for each available food f, which ingredients can be replaced by f. About 50 adaptation rules have been added to manage the foods available to cook a salad. For example, the four following adaptation rules:

$$\sigma_1 = (\texttt{SaladDish} \wedge \texttt{Diary} \rightsquigarrow \texttt{SaladDish} \wedge \texttt{Yogurt})$$
$$\sigma_2 = (\texttt{SaladDish} \wedge \texttt{CitrusFruit} \rightsquigarrow \texttt{SaladDish} \wedge \texttt{Orange})$$
$$\sigma_3 = (\texttt{SaladDish} \wedge \texttt{Egg} \rightsquigarrow \texttt{SaladDish} \wedge \texttt{Salmon})$$
$$\sigma_4 = (\texttt{SaladDish} \wedge \texttt{StoneFruit} \rightsquigarrow \texttt{SaladDish} \wedge \texttt{Strawberry})$$

state that, in salad recipes, dairy may be replaced by yogurt, citrus fruit may be replaced by orange, egg may be replaced by salmon, and stone fruit may be replaced by strawberry.

Let F be the set of available foods. The adaptation rules we defined are all of the form $\sigma = (\texttt{SaladDish} \wedge \texttt{A} \rightsquigarrow \texttt{SaladDish} \wedge \texttt{B})$, with $\texttt{B} \in \texttt{F}$. However, we can consider 3 types of rules relying on the relation between A and B:

- If $\texttt{B} \sqsubseteq \texttt{A}$ and $\texttt{A} \not\sqsubseteq \texttt{B}$ and $\nexists \texttt{C} \in \texttt{F}$ such that $\texttt{B} \neq \texttt{C}$ and $\texttt{C} \sqsubseteq \texttt{A}$, the adaptation rule allows to substitute all the foods more specific than A with B. For example, $\sigma = (\texttt{SaladDish} \wedge \texttt{Diary} \rightsquigarrow \texttt{SaladDish} \wedge \texttt{Yogurt})$ will allow to replace a dairy (e.g. Creme) appearing in a case base recipe by some yogurt. The constraint "$\nexists \texttt{C} \in \texttt{F}$ such as $\texttt{B} \neq \texttt{C}$ and $\texttt{C} \sqsubseteq \texttt{A}$" guarantees that there is no food in the fridge more specific than A other than B. For example, suppose that some Cheese (e.g. Cheddar) is available in the fridge, σ_1 will be split in more specific adaptatation rules, e.g. if Cheese and Creme are the only direct subclasses of Diary: $(\texttt{SaladDish} \wedge \texttt{Creme} \rightsquigarrow \texttt{SaladDish} \wedge \texttt{Yogurt})$, and $(\texttt{SaladDish} \wedge \texttt{Cheese} \rightsquigarrow \texttt{SaladDish} \wedge \texttt{Cheddar})$.
- If $\texttt{B} \not\sqsubseteq \texttt{A}$ and $\texttt{A} \not\sqsubseteq \texttt{B}$, the adaptation rule implies two foods which do not belong to a same category in the food hierarchy. This type of rule allows to take into account that two foods A and B play the same role in a salad dish and so, that they are substitutable. For example, σ_3 has been created because Egg and Salmon play the same role: they are proteins. This type of rule allows also to fix a food B as the closest available food of A. For example, σ_4 has been created because there is no stone fruit in the fridge and in this case, when StoneFruit appears in a source case recipe, the best way to substitute it with an available food is with Strawberry, StoneFruit and Strawberry being fruits.

- If $\texttt{B} = \texttt{Food}$, the rules state that there is no way to substitute A (e.g. Tea) appearing in a recipe of the case base with something available in the fridge. In this case, A is generalized into Food and an adaptation procedure is triggered to remove all the A which have been generalized into Food.

The impact of such rules in TUUURBINE is that new recipes are virtually created, because a recipe containing for example some grapefruit (which is a citrus fruit) will be retrieved as if this recipe contains some orange. So, the TUUURBINE retrieval process will be able to return recipes whose adaptation effort will be less costly because more controlled (the adaptation process remaining the same).

Let $\texttt{Q} = +\texttt{dishType:SaladDish} \wedge \texttt{ing:Salmon} \wedge \texttt{ing:Cucumber} \wedge \texttt{!Lemon:}$, be an example query meaning that the user wants a salad recipe with salmon and cucumber but without lemon. Consider the recipe named "Cucumber salad with hard-boiled egg and fromage blanc" which index is $idx(\texttt{R}) = \texttt{Egg} \wedge \texttt{FromageBlanc} \wedge \texttt{Cabbage} \wedge \texttt{Lemon} \wedge \texttt{Salt} \wedge \texttt{Pepper}$.

According to the fridge and to the food hierarchy, the unavailable foods are added to this initial query Q with a !. For example, FromageBlanc and Egg are unavailable classes of food. The extended query EQ submitted to TUUURBINE will be :

$$\texttt{EQ} = \texttt{Q} \wedge \texttt{!ing:FromageBlanc} \wedge \texttt{!ing:Egg} \wedge ...$$

For simplification, we present only the two classes of EQ which represent unavailable foods that will be taken into account by the adaptation rules. The first generalization which returns results is $\Gamma = \texttt{Cucumber} \rightsquigarrow \texttt{Vegetable}$ producing the generalized query: $\Gamma(\texttt{EQ}) = \texttt{dishType:SaladDish} \wedge \texttt{ing:Salmon} \wedge \texttt{ing:Vegetable} \wedge \texttt{!ing:Lemon} \wedge \texttt{!ing:FromageBlanc} \wedge \texttt{!ing:Egg}$. With the use of the three adaptation rules σ_1, σ_2 and σ_3, R matches $\Gamma(\texttt{EQ})$ because σ_1 transforms the FromageBlanc, a Diary of R, into Yogurt, σ_2 transforms the Lemon, a CitrusFruit of R, into Orange, σ_3 transforms the Egg of R into Salmon, and Cabbage is a Vegetable according to the food hierarchy. The answer returned by TUUURBINE for adapting R to Q consists in replacing Cabbage by Cucumber, Egg by Salmon, FromageBlanc by Yogurt and Lemon by Orange. The first subsitution comes from the basic retrieval/adaptation processes of TUUURBINE, while the three last ones come from the adaptation rules.

4.2 Adaptation of quantities for the salad challenge

The ontology-based substitution procedure extended by adaptation rules of TAAABLE favors the substitution of ingredients of the same type (a sauce by a sauce, a vegetable by a vegetable, etc.). So, ingredient quantities can, in most cases, be reused without adaptation. For example, 3 *cups* of Pasta can be replaced by 3 *cups* of Couscous, 1 *tsp* of Oregano by 1 *tsp* of Cumin, etc. However, there are some kinds of adaptations (coming from the adaptation rules) which require some quantity adjustements. An approach for the adaptation of ingredient quantities based on mixed linear optimization was proposed in [6] and has been used to compute sugar, alcohol and mass compensation when replacing some ingredients by others in the context of cocktail adaptations [5]. This complexity is not required to adapt salads, especially because

the substitution procedure is guided by adaptation rules. So, we only define a simple heuristic to provide realistic quantities to the user. This heuristic is the following. Each food available in the fridge is associated with a preferred unit and to a set of possible units coming from the recipes of the case base. For example, the set of possible units for `Carrot` is $\{unit, g, oz\}$ and the preferred unit is $unit$. For the quantity adaptation, if the replaced ingredient unit is a mass (e.g. g) or a volume (e.g. cl, tsp, $tblsp$) in the source case, and if this unit is a possible unit for the replacing ingredient, neither quantity adaptation nor unit adaptation is done. If not, conversion knowledge coming from WIKITAAABLE is used. First, the quantity of the replaced ingredient is converted into its mass, in grams. For example, if the source recipe uses 2 lettuces, the conversion knowledge stating that the mass of 1 lettuce is $360\ g$ is used to computed the total mass of lettuce: $2 \times 360\ g = 720\ g$). Secondly, conversion knowledge associates to each possible replacing ingredient (a food of the fridge) its preferred unit and the correspondance from this unit to a mass in grams. For example, the preferred unit of `Carrot` is $unit$ and the mass of $1\ unit$ of carrot is $70\ g$. This allows computing the adapted quantity in the preferred unit of the replacing ingredient, by divided the mass (in grams) of the replaced ingredient by the mass (in grams) corresponding to the preferred unit of the replacing ingredient. In the running example, if `Lettuce` is replaced by `Carrot`, and the original quantity of `Lettuce` is $2\ units$ then $720/70 \approx 10\ units$ of `Carrot` have to be used (the result is rounded).

5 Open Challenge: cocktail name adaptation

This section presents the TAAABLE team submission to the open challenge. The issue is, when TAAABLE returns an adapted recipe, how to name it according to its original name and to the substitution. For example, what name could be assigned to the recipe obtained from the "Green Russian" recipe after having substituted the mint liquor with curacao.

This issue of adapting recipe names has been suggested by the jury of the 2014 edition of the CCC and the CookingCAKE system [7] has addressed this challenge for the CCC-2015, using a few rules. For instance, the adjective `"cheesy"` is added to the recipe name if the adapted recipe of a sandwich contains some cheese. The TAAABLE team has addressed this issue more formally in [8], using an approach based on RDFS. In this application, a problem `pb` is a representation of a cocktail recipe by an RDFS graph. For the first version of this application, only ingredient types are considered, neither the quantities, nor the preparation steps. Therefore, a problem is an RDFS base $\texttt{pb} = \bigcup_{k=1}^{n} \{\langle id\ \texttt{ ing ?v}_k\rangle, \langle \texttt{?v}_k\ \texttt{ type }\ f_k\rangle\}$ where id is a constant (a resource identifying the recipe), $\texttt{?v}_1, \ldots, \texttt{?v}_n$ are n variables, and $f_1, \ldots, f_n$ are food classes.

A solution `sol(pb)` of `pb` is a literal of type string that gives a name to `pb`. It is assumed to be in lower case for the sake of simplicity, e.g., `sol(pb)` = `"green russian"` solves the problem `pb` represented in figure 1. The following operations on strings are used: concatenation (denoted by $+$, e.g., `"ab"` + `"cd"` = `"abcd"`), substring checking (denoted by `subStringOf`, e.g., `subStringOf("bc", "abcd")` = `true`), and string replacement (e.g., `replace("ab", "cd", "bababa")` = `"bcdcda"`).

A dependency $\beta_{\texttt{pb}}$ between pb and $\texttt{sol}(\texttt{pb})$ is an RDFS base. Usually, at least one food class f_k of pb and the literal $\texttt{sol}(\texttt{pb})$ occurs in $\beta_{\texttt{pb}}$: when it is not the case, $\beta_{\texttt{pb}}$ does not relate pb to $\texttt{sol}(\texttt{pb})$ (which is possible, e.g., when $\beta_{\texttt{pb}} = \emptyset$, i.e., there is no known dependency between pb and $\texttt{sol}(\texttt{pb})$). For each case $(\texttt{srce}, \texttt{sol}(\texttt{srce}))$, $\beta_{\texttt{srce}}$ is assumed to be given.

A matching $\alpha_{\texttt{pb}}$ from srce to tgt is either simple or complex. A simple matching has the form $f \rightsquigarrow g$ where f is a food class of srce and g is a food class of tgt; it represents the substitution of f by g. The removal of a food class f will be denoted by $f \rightsquigarrow \emptyset$. A complex matching is a composition $\alpha_{\texttt{pb}} = \alpha^q_{\texttt{pb}} \circ \alpha^{q-1}_{\texttt{pb}} \circ \ldots \circ \alpha^1_{\texttt{pb}}$ of simple matchings. $\alpha_{\texttt{pb}}$ is built during the adaptation of ingredients process of TAAABLE .

The matching α_β from $\beta_{\texttt{srce}}$ to $\beta_{\texttt{tgt}}$ is built during the cocktail name adaptation. It consists of a set of ordered pairs (d, d') where d is a descriptor of $\beta_{\texttt{srce}}$ and d' is a descriptor of $\beta_{\texttt{tgt}}$, a descriptor being either a resource (that can be a property) or a literal.

We present in the next sections five adaptation strategies: the two first strategies (§5.1 and §5.2) are application-dependent, whereas the last ones should be adaptable to other applications. Strategies presented in sections 5.3 and 5.4 are designed for simple matchings whereas the strategy of section 5.5 combines strategies for dealing with complex matchings.

5.1 Strategy "Alcohol abuse is dangerous for health"

Consider a cocktail recipe containing some alcohol, for which the adaptation consists in removing the ingredients which are alcohol or in substituting them by ingredients which are not alcohol. In this case, the new cocktail name may be computed by adding "virgin" to the original recipe name. For example, let $\texttt{sol}(\texttt{srce}) = \texttt{"mojito"}$ be the name of the "Mojito" recipe, let $\alpha_{\texttt{pb}} = \texttt{rhum} \rightsquigarrow \emptyset$ be the adaptation consisting in removing the unique alcohol of srce, then $\texttt{sol}(\texttt{tgt}) = \texttt{"virgin mojito"}$ will be the adapted cocktail name. Note that the test about alcohol can be performed by executing the SPARQL query $\texttt{Q}_{\texttt{alcohol}}$ (cf. equation (1)) twice:

- "srce contains some alcohol" is encoded by $\texttt{exec}_\vdash(\texttt{Q}_{\texttt{alcohol}}, \texttt{DK} \cup \texttt{srce}) \neq \emptyset$ and
- "tgt contains no alcohol" is encoded by $\texttt{exec}_\vdash(\texttt{Q}_{\texttt{alcohol}}, \texttt{DK} \cup \texttt{tgt}) = \emptyset$.

5.2 Default strategy

The default strategy is applied when all other strategies fail, this time by adding "the new " to the original recipe name (e.g. "the new bloody mary").

5.3 Strategy "Turn constants into variables"

This section models the adaptation example presented in Section 2, where the "Green Russian" recipe is adapted by replacing mint liquor by curacao.

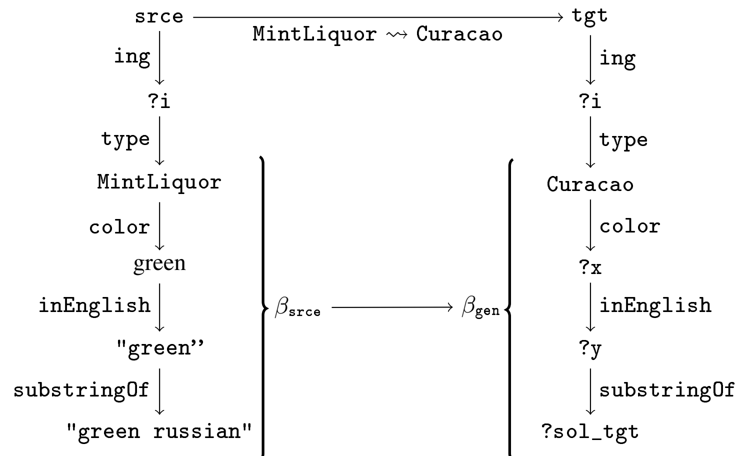


Fig. 4. Example of an RDFS based cocktail name adaptation using the "Turn constants into variables" strategy.

A partial explanation of the name $\mathtt{sol}(\mathtt{srce}) = \mathtt{"green\ russian"}$ is that the color of mint liquor is green, which can be modeled by:

$$\beta_{\mathtt{srce}} = \{\langle \mathtt{MintLiquor\ color\ green}\rangle, \langle \mathtt{green\ inEnglish\ "green"}\rangle, \langle \mathtt{"green"\ subStringOf\ "green\ russian"}\rangle\}$$

These triples are represented on the left-hand part of the graph of Fig. 4.

Since $\alpha_{\mathtt{pb}} = \mathtt{MintLiquor} \rightsquigarrow \mathtt{Curaçao}$, in order to build $\beta_{\mathtt{tgt}}$, the idea is to apply $\alpha_{\mathtt{pb}}$ on $\beta_{\mathtt{srce}}$ and then to make some modifications on the resources and literals to make it consistent with DK. This consistency test must be considered wrt CWA because there is no way to have $\langle \mathtt{MintLiquor\ color\ blue}\rangle$ inconsistent with DK in the classical semantics. It is assumed that $\mathtt{DK} \vdash_{\mathrm{cwa}} \neg\langle \mathtt{MintLiquor\ color\ blue}\rangle$, thus the mere substitution $\alpha_{\mathtt{pb}}$ on $\beta_{\mathtt{srce}}$ gives an inconsistent result wrt DK under CWA. So, the idea is to relax this triple. One way to do it is to replace green with a variable ?x. More generally, the strategy consists in replacing the descriptors of $\beta_{\mathtt{srce}}$ by variables, with the exception of the predicates (that are higher order resources) and of the descriptors occurring in tgt. The variable that replaces sol(srce) is ?solTgt: solving tgt consists in giving a value sol(tgt) to this variable. This gives the following dependency (obtained by applying $\alpha_{\mathtt{pb}}$ and turning some constants into variables):

$$\beta_{\mathtt{gen}} = \{\langle \mathtt{Curacao\ color\ ?x}\rangle, \langle \mathtt{?x\ inEnglish\ ?y}\rangle, \langle \mathtt{?y\ subStringOf\ ?solTgt}\rangle\}$$

$\beta_{\mathtt{gen}}$ is so-called, since it generalizes $\alpha_{\mathtt{pb}}(\beta_{\mathtt{srce}})$ (in the sense $\alpha_{\mathtt{pb}}(\beta_{\mathtt{srce}}) \vdash \beta_{\mathtt{gen}}$), where $\alpha_{\mathtt{pb}}(\beta_{\mathtt{srce}})$ is the result of applying the substitution $\alpha_{\mathtt{pb}}$ on $\beta_{\mathtt{srce}}$.

Now, in order to get $\beta_{\mathtt{tgt}}$, the idea is to unify the variables ?x and ?y with some constants, using the domain knowledge. Therefore DK is interrogated with the following SPARQL query: SELECT ?x ?y WHERE {Curacao color ?x . ?x inEnglish ?y}.

Assuming the only result is the pair $\{\texttt{?x} \leftarrow \texttt{blue}, \texttt{?y} \leftarrow \texttt{"blue"}\}$, it comes:

$$\beta_{\texttt{tgt}} = \{\langle \texttt{Curacao color blue}\rangle, \langle \texttt{blue inEnglish "blue"}\rangle, \langle \texttt{"blue" subStringOf ?solTgt}\rangle\}$$
$$\text{and } \alpha_\beta = \{(\texttt{MintLiquor}, \texttt{Curacao}), (\texttt{green}, \texttt{blue}), (\texttt{"green"}, \texttt{"blue"})\}$$

Therefore, $\beta_{\texttt{tgt}}$ involves that $\texttt{sol}(\texttt{tgt})$ has to respect the following constraint:

$$\texttt{sol}(\texttt{tgt}) \in \{s : \text{string} \mid \texttt{"blue"} \text{ is a substring of } s\} \tag{5}$$

Now, $\texttt{sol}(\texttt{srce})$ must be modified using α_β into $\texttt{sol}(\texttt{tgt})$ that respects (5). Here, a domain-dependent choice is made: it concerns the way the solution space is structured, i.e., how can modifications be applied on solutions. It is assumed that in this application, the only modification operation is based on the `replace` operation on the set of strings (which is the solution space). Hence, since $(\texttt{"green"}, \texttt{"blue"}) \in \alpha_\beta$, the following cocktail name that is consistent with (5) is proposed:

$$\texttt{sol}(\texttt{tgt}) = \texttt{replace}(\texttt{"green"}, \texttt{"blue"}, \texttt{sol}(\texttt{srce})) = \texttt{"blue russian"}$$

5.4 Strategy "Generalization-specialization of dependencies"

Now, consider the example of the adaptation of $\texttt{sol}(\texttt{srce}) = \texttt{"green russian"}$ when $\alpha_\beta = \texttt{MintLiquor} \rightsquigarrow \texttt{IndianTonic}$ with the same $\beta_{\texttt{srce}}$ as in section 5.3 and assuming that DK gives no color to Indian tonic (i.e., there is no triple of the form $t = \langle \texttt{IndianTonic color c}\rangle$ such that $\texttt{DK} \vdash t$), the adaptation strategy of section 5.3 fails. However, it is assumed that

$$\texttt{DK} \vdash \left\{\begin{array}{c} \langle \texttt{IndianTonic taste bitter}\rangle, \langle \texttt{IndianTonic texture sparkling}\rangle, \\ \langle \texttt{bitter inEnglish "bitter"}\rangle, \langle \texttt{sparkling inEnglish "sparkling"}\rangle, \\ \langle \texttt{color subp hOP}\rangle, \langle \texttt{taste subp hOP}\rangle, \langle \texttt{texture subp hOP}\rangle \end{array}\right\}$$

meaning that Indian tonic is bitter and sparkling, and that color, taste and texture are organoleptic properties (`hOP` is an abbreviation for `hasOrganolepticProperty`). Therefore, the adaptation strategy described in section 5.3 can be applied with a slight modification: it is sufficient to replace in $\beta_{\texttt{gen}}$ the triple $\langle \texttt{IndianTonic color ?x}\rangle$ by $\langle \texttt{IndianTonic hOP ?x}\rangle$, which is more general according to DK.

One way to address this problem is to search in the domain knowledge for triples for building $\beta_{\texttt{gen}}$ that are *similar* to $\alpha_\beta(\beta_{\texttt{srce}})$. This can be likened to the retrieval issue in CBR, which can be implemented by a least generalization of the query (see, e.g., [9]). A similar idea is proposed here. It consists in making a best-first search in a space of dependencies β such that:

- The initial state β_0 corresponds to the $\beta_{\texttt{gen}}$ as it is computed in the strategy of section 5.3.
- The successors of a state consists in making a generalization of one of its triples. The following generalization operators can be considered: replace a class (resp., a property) by a direct superclass (resp., direct superproperty) in DK, replace a resource or a literal by a variable, etc. A cost function must be associated to generalization operators, in order to choose the least costly generalization.

– A final state β is such that the SPARQL query associated with it gives a nonempty set of results.

Once a final state β is found, the rest of the approach of Section 5.3 can be applied with $\beta_{\texttt{gen}} = \beta$.

Back to the example, it comes:

$$\beta_0 = \{\langle \texttt{IndianTonic color ?x} \rangle, \langle \texttt{?x inEnglish ?y} \rangle,$$
$$\langle \texttt{?y subStringOf ?solTgt} \rangle\}$$

In the first triple, `color` can be generalized into `hOP` (since $\texttt{DK} \vdash \langle \texttt{color subp hOP} \rangle$), giving

$$\beta = \{\langle \texttt{IndianTonic hOP ?x} \rangle, \langle \texttt{?x inEnglish ?y} \rangle,$$
$$\langle \texttt{?y subStringOf ?solTgt} \rangle\}$$

β is a final state since $\texttt{exec}_{\vdash}(\texttt{Q}, \texttt{DK}) \neq \emptyset$ for

$$\texttt{Q} = \text{SELECT ?x ?y WHERE } \{\texttt{IndianTonic hOP ?x . ?x inEnglish ?y}\}$$

Indeed, $\texttt{exec}_{\vdash}(\texttt{Q}, \texttt{DK}) = \{A_1, A_2\}$ where $A_1 = \{\texttt{?x} \leftarrow \texttt{bitter}, \texttt{?y} \leftarrow \texttt{"bitter"}\}$ and $A_2 = \{\texttt{?x} \leftarrow \texttt{sparkling}, \texttt{?y} \leftarrow \texttt{"sparkling"}\}$, leading to the two expected solutions: `"bitter russian"` and `"sparkling russian"`.

Therefore this strategy consists in finding the minimal generalization β of the initial dependency β_0 and then in specializing β into $\beta_{\texttt{tgt}}$'s thanks to SPARQL querying on `DK`, hence the name of the strategy.

5.5 Composing strategies when the matching is complex

When the matching $\alpha_{\texttt{pb}}$ is complex, it can be written $\alpha_{\texttt{pb}} = \alpha_{\texttt{pb}}^q \circ \alpha_{\texttt{pb}}^{q-1} \circ \ldots \circ \alpha_{\texttt{pb}}^1$, with $q \geq 2$. The idea is then to apply in sequence the strategies associated with simple matchings. For example, for $\texttt{sol}(\texttt{srce}) = \texttt{"green russian"}$, $\alpha_{\texttt{pb}}^1 = \texttt{mintLiquor} \rightsquigarrow \texttt{curacao}$, $\alpha_{\texttt{pb}}^2 = \texttt{vodka} \rightsquigarrow \texttt{tequila}$, the strategy of Section 5.3 can be applied twice to give the name $\texttt{sol}(\texttt{tgt}) = \texttt{"blue mexican"}$. This adaptation is an application of the adaptation based on reformulations and similarity paths (see e.g. [10]).

6 Conclusion

This paper has presented the systems developed by the TAAABLE team for its participation to the 2017 CCC. The two systems presented for the *salad* and *mixology* challenges are based on the successful 2015 version of TAAABLE , extended for *salad* challenge with a new approach to manage the fridge. A new approach has also been presented for adapting the cocktail names from the ingredient adaptation. Several name adaptation strategies have been presented and, if some proposed strategies are application-dependent, it is claimed that other ones can be applied—or adapted—to a larger framework. Indeed, they match the principles described in some related work about analogical

transfer (e.g., [11] and [12]) while proposing an approach benefitting from the standard RDFS and associated tools (RDFS SPARQL engines, RDF stores). A first prototype implementing the three first strategies has already been developed, but the adaptation strategy based on generalization-specialization of dependencies is under development. However, there is an important workload for acquiring dependencies $\beta_{\texttt{srce}}$, which is currently done manually and for acquiring triples in the domain knowledge. A possibility to address these issues is to query the Linked Open Data (LOD), a huge cloud of RDF and RDFS bases freely accessible on the Web. This knowledge acquisition task is the main future work.

References

1. E. Gaillard, L. Infante-Blanco, J. Lieber, and E. Nauer. Tuuurbine: A Generic CBR Engine over RDFS. In *Case-Based Reasoning Research and Development*, volume 8765, pages 140 – 154, Cork, Ireland, September 2014.
2. A. Cordier, V. Dufour-Lussier, J. Lieber, E. Nauer, F. Badra, J. Cojan, E. Gaillard, L. Infante-Blanco, P. Molli, A. Napoli, and H. Skaf-Molli. Taaable: a Case-Based System for personalized Cooking. In S. Montani and L. C. Jain, editors, *Successful Case-based Reasoning Applications-2*, volume 494 of *Studies in Computational Intelligence*, pages 121–162. Springer, 2014.
3. E. Gaillard, J. Lieber, and E. Nauer. Adaptation knowledge discovery for cooking using closed itemset extraction. In *The Eighth International Conference on Concept Lattices and their Applications - CLA 2011*, pages 87–99, 2011.
4. B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, Berlin, 1999.
5. E. Gaillard, J. Lieber, and E. Nauer. Improving Ingredient Substitution using Formal Concept Analysis and Adaptation of Ingredient Quantities with Mixed Linear Optimization. In *Computer Cooking Contest Workshop*, Frankfort, Germany, September 2015.
6. J. Cojan and J. Lieber. Applying Belief Revision to Case-Based Reasoning. In *Computational Approaches to Analogical Reasoning: Current Trends*, volume 548 of *Studies in Computational Intelligence*, pages 133 – 161. Springer, 2014.
7. G. Müller and R. Bergmann. CookingCAKE: A Framework for the adaptation of cooking recipes represented as workflows. 2015.
8. N. Kiani, J. Lieber, E. Nauer, and J. Schneider. . In A. K. Goel et al., editor, *Case-Based Reasoning Research and Development, Proceedings of the 24th International Conference on Case-Based Reasoning (ICCBR-2016)*, Lecture Notes in Computer Science, Atlanta, Georgia, 2016. Springer International Publishing.
9. E. Gaillard, L. Infante-Blanco, J. Lieber, and E. Nauer. Tuuurbine: A Generic CBR Engine over RDFS. In *Case-Based Reasoning Research and Development*, volume 8765, pages 140 – 154, Cork, Ireland, September 2014.
10. E. Melis, J. Lieber, and A. Napoli. Reformulation in Case-Based Reasoning. In B. Smyth and P. Cunningham, editors, *Fourth European Workshop on Case-Based Reasoning, EWCBR-98*, LNCS 1488, pages 172–183. Springer, 1998.
11. D. Gentner. Structure-Mapping: A Theoretical Framework for Analogy. *Cognitive science*, 7(2):155–170, 1983.
12. P. H. Winston. Learning and reasoning by analogy. *Communications of the ACM*, 23(12):689–703, 1980.