

Extending answers to neighbour entities in a Cooperative Answering context *

F.Cuppens R.Demolombe

ONERA-CERT
2 Av. E.Belin, BP3025
Toulouse Cedex
France

June 3, 2002

Abstract

When a user has to retrieve information in a Database, a standard DBMS provides him just the exact answers to his queries. In a Cooperative Answering context a system has to provide him additional interesting information. In this paper we present a method to determine the interesting information from a given query and a knowledge base which represents the expertise of an expert in providing information. In this knowledge base are represented rules defining neighbour entities. These entities are obtained by extending the query to neighbour entity types or to neighbour conditions. The knowledge base is formalized in First Order Logic with two language levels: object level and meta level. The concept of neighbourhood between predicates or conditions is expressed at the meta level.

1 Introduction

In the context of Advice Giving Systems or Decision Support Systems, if a user has to access data in a Database, a standard Relational DBMS is not well appropriate because it provides no more information than the precise answer to a given query, and in many cases users have not a very precise idea of the information in the Database that could help them to solve a particular problem.

That is why a more flexible system which can provide additional interesting information is more suitable in this context. In [4], we have presented a general methodology, called Cooperative Answering, to design such a system, where answers are entities with some of their attribute values. In this approach, there are two kinds of additional information:

*This work was supported by the ESPRIT project: ESTEAM-316

additional attributes for the entities requested by the query, or additional entities satisfying less restrictive conditions. We have shown in [5] how the concept of user's topic of interest allows to define the interesting additional attributes. The goal of this paper is to present a method to determine the interesting additional entities.

The following examples give an intuitive idea of what could be these additional entities. Let's consider a user who wants to plan a travel and asks the query:

What is the departure time and the price of the flights from Paris to New York, whose departure time is between 8a.m and 12a.m ?

A standard Relational DBMS would supply the answer:

Flight	Departure-time	Price
AF001	11h00	27,715FF
AF015	10h30	8,825FF

However an employee in a Travel Agency, having experience in giving advices to plan a travel, knows that some people accept to "slightly" change the departure city or the arrival city, if the change of the total distance is not too big, or to "slightly" change the departure time. There may be different reasons to accept these changes, for example to get a less expensive flight. That is why this employee might propose the following *additional* flights:

Flight	Departure-time	Price	Departure-city	Arrival-city
AF054	12h10	8,500FF	Paris	New York
AF020	09h30	7,850FF	Paris	Washington
BA017	09h00	6,750FF	London	New York
BA142	10h10	6,500FF	London	Boston

A consequence of these changes is the necessity to give the user the departure city and the arrival city, though they are not requested in the query, because he has to know the conditions which have been changed. The reason to give these additional attribute values is very different than the reason presented in [5]. Indeed, in [5], additional attributes correspond to interesting topics or to exceptions with respect to some common sense rules.

Let's consider now the query:

What is the departure time of the flights from Paris to Brussels, whose departure time is between 7a.m and 11a.m ?

The answer provided by an expert in a Travel Agency would be (it is not the exact answer since the price is provided although it is not explicitly requested by the user):

Flight	Departure-time	Price
AF638	08h05	1,850FF
AF642	09h25	1,850FF

In that case, it is sensible to relax the condition about the departure time like for the previous example, but it would not be sensible to change the conditions about the cities because the distance between Paris and Brussels is relatively small. Then, the expert might also propose the following additional answers:

Flight	Departure-time	Price
S651	11h07	1,620FF

Train	Departure-time	Price
SNCF281	07h48	450FF
SNCF483	10h08	450FF

The reason is that he knows that the distance between the two cities is not so important (less than 400km), and that trains offers some advantages (for example they are cheaper and it is not necessary to go to an airport). Although it is not explicitly requested by the user, the attribute *Price* is inserted in the answer to give the user the possibility to compare the advantage of the trains with respect to the price. The implicit reasoning of the expert is that flights and trains are two particular means of travel, which can be interchanged in some contexts depending on the distance.

These two examples show two possibilities to give additional entities: to relax some conditions on the attributes in the query, or to consider entities of a “similar” type. Here “similar” means having the same father in some entity type hierarchy, plus other conditions.

Other works in the same stream can be found in the literature, in particular in [7] and [9]. The reason advocated in [7] to provide cooperative answers concerns empty answers; it can be noticed that our method is not limited to these situations. A more precise comparison with the methods defined in these papers is presented in the conclusion. The most important difference, in our view, is that it is formalized in First Order Logic.

In the next sections we have first to recall part of the background presented in [4] and in [2]. Then we introduce the concept of neighbourhood between conditions and between entity types, and the rules defining the contexts in which they can be applied. Farther off are presented the rules which transform a given query into other queries whose answers provide the additional information. The last section shows how it is possible to restrict the set of neighbour entities with respect to an optimisation criterium.

2 Database structure representation

To define a query transformation we need a formalism to represent the queries and the database structure (the “database scheme”, in database

terminology). The formalism we have selected is First Order Logic (FOL), and we distinguish two levels: object level and meta level (see [1] and [6]). The reasons to use two levels are that the transformations concern the meaning (their intention) of the queries, and not the entities they refer (their extension), and also because the concepts of the Entity-Relationship model, which are more convenient to define the method, cannot be directly expressed in a one level FOL.

2.1 Object level language

The object level language is a First Order Language where predicate symbols are those of the Relational schema of a given application, plus unary predicates to represent the relation domains.

For example the predicate symbols may be:

$$\begin{aligned} & \textit{Departure-time}(x,y), \textit{Departure-city}(x,y), \textit{Arrival-city}(x,y)... \\ & \textit{Means-of-Travel}(x), \textit{Flight}(x), \textit{Train}(x)... \end{aligned}$$

The query language is a restriction of this language to formulas of the form:

$$\textit{Entity} \wedge \textit{Condition} \wedge \textit{Retrieved Attributes}$$

where “*Entity*” is a formula with only entity type predicates and with the only logical operators \wedge and \vee ; “*Condition*” is whatever formula whose free variables are free variables of “*Entity*” or “*Retrieved Attributes*”; “*Retrieved Attributes*” is a conjunction of positive atomic formulas.

The first query is expressed in this language by the formula:

$$\begin{aligned} & \textit{Flight}(x) \wedge \\ & (\textit{Departure-city}(x,Paris) \wedge \textit{Arrival-city}(x,New York) \wedge \\ & \textit{Departure-time}(x,y) \wedge (8 < y) \wedge (y < 12)) \wedge \\ & (\textit{Departure-time}(x,y) \wedge \textit{Price}(x,z)) \end{aligned}$$

The intuition behind this particular form is that a query expresses that a user wants to know the value of the attributes in the Retrieve Attribute part, for the entities having a type defined by the Entity part and satisfying the Condition part.

These restrictions make more easy the transformation definition.

2.2 Meta level language

The meta level language is a First Order Language whose predicate symbols are defined below. The predicates and the formulas of the object language are represented at this level by constants. These constants are the result of a coding function which assigns a code to any symbol or formula of the object language [1]. To understand more easily the meaning of these constants, the code of a formula is denoted by the formula itself between quotes.

For example:

$$\textit{“Flight}(x) \wedge \textit{Departure-city}(x, Paris)”}$$

denotes a constant which is the code of the formula inside the quotes; note that at this level the object variable x has the status of a constant. In some cases we can have meta variables to denote any object formula of a particular form. For example if v is a meta variable:

$$"Flight(x) \wedge Departure-city(x, < v >)"$$

is a term, and $< v >$ denotes that v has not the same status as x .

In the following when there is no risk of misunderstanding, the quotes and $< >$ are omitted.

The structure of the Database is represented using the concepts of: Entity type, Attribute of an entity, Association, Attribute of an association, and Entity type structure.

We have for example the meta predicates:

Entity-type(x): x is an entity type.

Att(x,y): x is an attribute of an entity of type y .

ISA(x,y): x is a special case of the entity type y .

They allow for example to define the structure:

$$\begin{aligned} &Entity-type(Flight), Entity-type(Train), Entity-type(Means-of-Travel)... \\ &Att(Departure-time, Means-of-Travel), \\ &Att(Departure-city, Means-of-Travel)... \\ &ISA(Flight, Means-of-Travel), ISA(Train, Means-of-Travel)... \end{aligned}$$

There are also axioms to express that *ISA* is a reflexive and transitive predicate, and that attributes are inherited in the *ISA* hierarchy of the entity types. See [5] for a presentation and a justification of these axioms, and for results about completeness and soundness relating axioms at the meta level and at the object level.

Only the meta predicates needed in this paper are presented here; a more detailed presentation can be found in [2].

The queries are represented with the meta predicates:

$$Entity(x,y), Condition(x,y), Ret-Att(x,y)$$

where x is the code of a query and y is the code of its Entity part (resp. Condition part, Retrieved Attribute part).

For example the first query is represented by the formulas:

$$Entity(qt1, "Flight(x)")$$

$$\begin{aligned} Condition(qt1, &"Departure-city(x, Paris) \wedge \\ &Arrival-City(x, New York) \wedge \\ &Departure-time(x,y) \wedge (8 < y) \wedge (y < 12)") \end{aligned}$$

$$Ret-Att(qt1, "Departure-time(x,y) \wedge Price(x,z)")$$

The formal definition of the answers is defined as usual (see for example [8]). In [3] we have shown that it is more meaningful to present the answers

in the form of instantiated formulas than in the form of tuples. In the example in the Annex answers are represented in this form.

The next sections are structured in such a way that the analogy between neighbour entity types and neighbour condition is clearly pointed out.

3 Characterization of neighbour entities

3.1 Neighbour entity types

We introduce the meta predicate:

$$Neighbour-Ent(q, E_1(x), E_2(x))$$

to express that for the query q the entity type E_1 may be replaced by the entity type E_2 .

With this predicate it is possible to represent expert's knowledge relative to a particular application domain by rules like:

$$\begin{aligned} \text{R1: } \forall (q, x, v_1, v_2) \\ & entity(q, "Flight(x)") \wedge \\ & condition(q, "Departure-city(x, v_1) \wedge Arrival-city(x, v_2)") \wedge \\ & Distance(v_1, v_2) < 400km \\ & \longrightarrow Neighbour-Ent(q, "Flight(x)", "Train(x)") \end{aligned}$$

This rule expresses that for any query q about entities of type *Flight*, with a condition about the *departure city* and the *arrival city* such that their distance is less than 400km, it is relevant to transform the query by replacing the entity type *Flight* by *Train*.

It is easy to see with this example that an entity type transformation based only on the ISA structure would lead to irrelevant and stupid transformations; for example to request Trains to go from Paris to New-York !

The antecedent part of this kind of rule allows to express in which particular context an entity type can be considered to be neighbour with another one. Of course, it is the role of experts to define this context.

It must be noticed that in the antecedent part some premises have to be proved at the meta level while other ones, like: $Distance(v_1, v_2) < 400km$, have to be proved at the object level (see [1] for a study of the theoretic problems arising in the amalgamation of language and meta language).

There is also a rule R2, which is independent of a particular application domain, to express that the property of neighbourhood can be inherited to more specific entity types.

$$\begin{aligned} \text{R2: } \forall (q, e_1, e_2, e'_1, x) \\ & Neighbour-Ent(q, "e_1(x)", "e_2(x)") \wedge ISA(e'_1, e_1) \\ & \longrightarrow Neighbour-Ent(q, "e'_1(x)", "e_2(x)") \end{aligned}$$

This rule says that if entity types e_1 and e_2 are neighbour and e'_1 is a special case of e_1 then e'_1 and e_2 are neighbour.

3.2 Neighbour conditions

We introduce the meta predicate:

$$\text{Neighbour-Cond}(q, E(x), C_1, C_2)$$

to express that for the query q about entities of type E , where the Condition part “contains” the condition C_1 , this condition can be replaced by the condition C_2 .

This predicate allows to represent knowledge like:

$$\begin{aligned} \text{R3: } \forall (q, x, v_1, v_2) \\ & \text{entity}(q, \text{“Means-of-Travel}(x)\text{”}) \wedge \\ & \text{condition}(q, \text{“Departure-city}(x, v_1) \wedge \text{Arrival-city}(x, v_2)\text{”}) \\ & \longrightarrow \text{Neighbour-Cond}(q, \text{“Means-of-Travel}(x)\text{”}, \\ & \qquad \qquad \qquad \text{“Departure-city}(x, v_1) \wedge \text{Arrival-city}(x, v_2)\text{”}, \\ & \qquad \qquad \qquad \text{“Departure-city}(x, v'_1) \wedge \text{Arrival-city}(x, v'_2)\text{”}) \wedge \\ & \qquad \qquad \qquad \text{Neighbour-Travel}(v'_1, v_1, v'_2, v_2) \end{aligned}$$

This rule expresses that for any query q about the entities *Means-of-Travel*, with a condition on *departure city* and *arrival city*, a neighbour condition can be obtained by replacing these two cities by two other “neighbour” cities. The “neighbour” cities are defined by the object level predicate *Neighbour-Travel*.

Another example of such a rule which transforms the conditions on departure time is:

$$\begin{aligned} \text{R4: } \forall (q, x, h) \\ & \text{entity}(q, \text{“Means-of-Travel}(x)\text{”}) \wedge \\ & \text{condition}(q, \text{“Departure-time}(x, h)\text{”}) \\ & \longrightarrow \text{Neighbour-Cond}(q, \text{“Means-of-Travel}(x)\text{”}, \\ & \qquad \qquad \qquad \text{“Departure-time}(x, h)\text{”}, \\ & \qquad \qquad \qquad \text{“Departure-time}(x, h') \wedge |h' - h| < 15mn\text{”}) \end{aligned}$$

In the rules R3 and R4 the neighbour condition corresponds to a condition where the constants (cities or departure time) are replaced by neighbour constants. These neighbour constants are defined by an explicit set, in the case of cities, or by an implicit set, in the case of departure time.

We can also use the concept of neighbourhood to represent the fact that two predicates express neighbour conditions. For example, if we have the predicates *First-Class-Price*(x, y) and *Business-Class-Price*(x, y) to represent the ticket price in first class or in business class, the next rule expresses that they impose neighbour conditions :

$$\begin{aligned} \text{R5: } \forall (q, x, y) \\ & \text{entity}(q, \text{“Flight}(x)\text{”}) \\ & \text{condition}(q, \text{“First-Class-Price}(x, y)\text{”}) \\ & \longrightarrow \text{Neighbour-Cond}(q, \text{“Flight}(x)\text{”}, \\ & \qquad \qquad \qquad \text{“First-Class-Price}(x, y)\text{”}, \\ & \qquad \qquad \qquad \text{“Business-Class-Price}(x, y)\text{”}) \end{aligned}$$

This rule can be used for the query :

What is the flight company for the flights from Paris to New-York such that the first class ticket price is less than 10,000FF ?

in order to replace the condition on a first class price by a condition on a business class price. It can be noticed that this kind of neighbourhood cannot be defined with the method presented in [7].

The rules R3, R4 and R5 are specific to a given application domain. The next rule R6 is general, and expresses that the property of neighbourhood is inherited to more specific entity types.

$$\begin{aligned} \text{R6: } \forall (q, c_1, c_2, e, e', x) \\ \text{Neighbour-Cond}(q, "e(x)", c_1, c_2) \wedge \text{ISA}(e', e) \\ \longrightarrow \text{Neighbour-Cond}(q, "e'(x)", c_1, c_2) \end{aligned}$$

All the rules presented in this section allow to derive, in a given application and for a given query, what are the entities neighbouring those explicitly requested by the user.

In this approach the notion of neighbourhood is defined by semantic knowledge contrary to other approaches based on general abstract computation rules, like in fuzzy sets.

4 Query transformation

In this section we define a query transformation which uses the information derived about the predicates *Neighbour-Ent* and *Neighbour-Cond*.

Roughly speaking in the previous section we have shown how to determine the interesting additional information. In this section we define how this information can be effectively obtained.

The transformed queries are defined with three meta predicates representing the transformation of the three parts of the query. We will not explicit the general rules which compound the transformation result of each part to build the global transformed queries.

4.1 Neighbour entity types

To transform the entity part we have the meta predicate:

$$\text{Transf-Ent}(q, e_1, e_2, e, e')$$

where: e is the entity part of the query q , e_1 is an atomic entity type which appears in e , e_2 is the atomic entity type which can replace e_1 in e , and e' is the entity part of the transformed query.

For example if in a query the entity part e is: $\text{Flight}(x) \vee \text{Train}(x)$, the atomic entity type e_1 : $\text{Train}(x)$, could be replaced by e_2 : $\text{Car}(x)$, and e' would be: $\text{Flight}(x) \vee \text{Car}(x)$.

The meta predicate:

$$\text{entity}(q, e)$$

expresses that e is an atomic entity type in the query q .

The next rule R7 says that if for the query q the entity type e_1 is neighbour with e_2 , and e_2 doesn't appear in q , then the entity part in the transformed query is e' obtained by replacing in e the subformula e_1 by e_2 .

$$\begin{aligned}
\text{R7: } & \forall (q, e_1, e_2, e, e') \\
& \text{Entity}(q, e) \wedge \\
& \text{Neighbour-Ent}(q, e_1, e_2) \wedge \\
& \neg \text{entity}(q, e_2) \wedge \\
& \text{Substitute}(e_1, e_2, e, e') \\
& \longrightarrow \text{Transf-Ent}(q, e_1, e_2, e, e')
\end{aligned}$$

Sometimes, when the entity type is transformed, the conditions have to be adapted to the new entity type. This is represented with the new meta predicate:

$$\text{Transf1-Cond}(q, e_1, e_2, c, c')$$

where c' is the transformed condition part when the entity type e_1 is replaced by e_2 in the query q .

The reason why this change is done may be the fact that the attributes which appear in c are not all defined for the entity type e_2 , or because by nature of the entity type e_2 the conditions have to be modified. For example if *Flight* are replaced by *Train* the condition on the *departure time* could be replaced by the same condition on the *arrival time*, because the duration of the travel is significantly longer with a train. Then we may have a rule like R8:

$$\begin{aligned}
\text{R8: } & \forall (q, c, x, y) \\
& \text{condition}(q, \text{"Departure-time}(x, y)") \wedge \\
& \text{Neighbour-Ent}(q, \text{"Flight}(x)", \text{"Train}(x)") \\
& \longrightarrow \text{Transf1-Cond}(q, \text{"Flight}(x)", \text{"Train}(x)", \\
& \qquad \qquad \qquad \text{"Departure-time}(x, y)", \\
& \qquad \qquad \qquad \text{"Arrival-time}(x, y)")
\end{aligned}$$

4.2 Neighbour conditions

To transform the condition part of a query we have the meta predicate:

$$\text{Transf2-Cond}(q, c_1, c_2, c, c')$$

where c_1 is a condition in the condition part c of q , c_2 is a condition which can be substituted to c_1 , and c' is the resulting condition part.

The facts derived about the predicate *Neighbour-Cond* are used in the rule R9 to derive the corresponding transformed query:

$$\begin{aligned}
\text{R9: } & \forall (q, e, c, c', c_1, c_2) \\
& \text{Condition}(q, c) \wedge \\
& \text{Neighbour-Cond}(q, e, c_1, c_2) \wedge \\
& \text{Rel-for-All}(q, e) \wedge \\
& \text{Substitute}(c_1, c_2, c, c') \\
& \longrightarrow \text{Transf2-Cond}(q, c_1, c_2, c, c')
\end{aligned}$$

The *Substitute* predicate is used to replace in c , the subformula c_1 by c_2 to obtain c' .

The *Rel-for-all* predicate is defined by:

$$\text{DEF: } \forall (q, e, x) \\ \text{Rel-for-All}(q, "e(x)") \longleftrightarrow (\forall(e') \text{entity}(q, "e'(x)") \longrightarrow \text{ISA}(e', e))$$

This predicate expresses that all the atomic entity formulas in the entity part of q are special cases of a given entity type e . With this predicate the transformation defined by the rule R9 is restricted to the cases where the neighbourhood between c_1 and c_2 is valid for all the atomic entity types in q .

For example if c_1 is neighbour of c_2 only for entities of type *Flight*, and the entity part of q is: $\text{Flight}(x) \vee \text{Train}(x)$, the transformation R9 cannot be applied. Nevertheless if the entity part is: $\text{TWA-Flight}(x) \vee \text{JAL-Flight}(x)$, and these two entity types are special cases of *Flight*, then the transformation can be applied.

The rule R10 defines the transformation in the case where *Rel-for-all* doesn't hold. In this case the transformed condition is applied only to entities such that *Neighbour-Cond* holds. This remark leads to substitute " $e \wedge c_2$ " to c_1 , instead of c_2 .

$$\text{R10: } \forall (q, e, c, c', c_1, c_2) \\ \text{Condition}(q, c) \wedge \\ \text{Neighbour-Cond}(q, e, c_1, c_2) \wedge \\ \text{entity}(q, e) \wedge \\ \neg \text{Rel-for-All}(q, e) \wedge \\ \text{Substitute}(c_1, "e \wedge c_2", c, c') \\ \longrightarrow \text{Transf2-Cond}(q, c_1, c_2, c, c')$$

When entities satisfying neighbourhood conditions are provided, we have also to provide the value of modified conditions, because the user does not know how they have been changed.

For example, we have to provide the value of *Departure-time* and *Arrival-time* attribute when the rule R3 is used, and the value of *Departure-time* for the rule R4.

To provide this information, we introduce the meta-predicate:

$$\text{Transf2-Ret-Att}(q, c_1, c_2, I)$$

I is an interesting information to be provided in neighbour answers when the condition c_1 is modified by c_2 .

The rule R11 defines the attribute values to be provided as a side effect of condition changes. The meta predicate *Info-Int* uses only syntactical criteria to recognize interesting attributes (actually, I is built with all the attribute predicates which appear in c_2)

$$\text{R11: } \forall (q, e, c_1, c_2, I) \\ \text{Neighbour-Cond}(q, e, c_1, c_2) \wedge \\ \text{Info-Int}(c_2, I) \\ \longrightarrow \text{Transf2-Ret-Att}(q, c_1, c_2, \text{True}, I)$$

About the condition transformation, we have also to make the following important remark:

Let's assume that we can deduce the two following theorems:

$$\textit{Neighbour-Cond}(q, s, c_1, c'_1) \text{ and } \textit{Neighbour-Cond}(q, s, c_2, c'_2)$$

and that we have a query q with an entity part ϵ and a condition part $C = c_1 \wedge c_2$.

From R9, we can derive:

$$\textit{Transf2-Cond}(q, c_1, c'_1, C, c'_1 \wedge c_2) \text{ and } \textit{Transf2-Cond}(q, c_2, c'_2, C, c_1 \wedge c'_2)$$

and we will generate two transform queries q_1 and q_2 having respectively for condition part: $C_1 = c'_1 \wedge c_2$ and $C_2 = c_1 \wedge c'_2$. This shows that in transformed queries, we change one condition at a time.

Actually, we could have generated only one query q' with the following condition part: $C' = c'_1 \wedge c'_2$.

We have not chosen this solution, because we consider that $c'_1 \wedge c_2$ and $c_1 \wedge c'_2$ formulas are closer to $c_1 \wedge c_2$ than $c'_1 \wedge c'_2$.

Of course, we do not assume that the neighbourhood relation is transitive, but the answers to the query q' could be obtained by applying the condition transformation process to the queries q_1 and q_2 . Indeed, we would then obtain three queries having respectively the condition part $c''_1 \wedge c_2$, $c_1 \wedge c''_2$ and $c'_1 \wedge c'_2$ with:

$$\textit{Neighbour-Cond}(q, s, c'_1, c''_1) \text{ and } \textit{Neighbour-Cond}(q, s, c'_2, c''_2)$$

These queries are neighbour with neighbours of q .

Thus, we can introduce a concept of neighbourhood degree which allows to obtain more and more information by propagating the neighbourhood notion. This propagation has to be controlled by the user who can stop the process when he has obtained what he wants.

5 Optimisation criteria to restrict neighbour entities

In many cases, neighbour entities are interesting or not depending on the value of some criterium which is chosen to compare entities. These criteria strongly depend on the application domain, and on each particular context. In particular they may depend on the user category.

For example for some user category an optimisation criterium may be the price of a travel, for other ones it may be the duration.

The idea is to avoid to provide neighbour entities having a criterium value greater than the lowest value for the answers of the initial query (in the case we want to minimize this criterium).

In this section we present meta predicates to represent this notion of optimisation criterium and corresponding rules.

To introduce a new optimization criterium, we use the meta predicate:

Opt-Crit(q, E(x), C)

to express that for the query q about entities of type E , C is an optimization criterium formula.

For example, the rule R12 says that for *Means-of-Travel* entities, the *Price* may be an optimization criterium. So, if the entity part of the initial query q concerns entities of type E , neighbourhood entities of type *Means-of-Travel* will be provided if their price is less than the price of any initial solutions (characterized by the formula $E(x') \wedge C'$).

The formula C' is a variant of the condition part C where free variables $X = x_1, \dots, x_n$ of C have been renamed as $X' = x'_1, \dots, x'_n$.

$$\begin{aligned} \text{R12: } \forall (q, x, E, C, C', X') \\ & \text{entity}(q, "E(x)") \wedge \text{condition}(q, C) \wedge \text{Variant}(C, C', X') \\ & \longrightarrow \text{Opt-Criterium}(q, "Means-of-Travel(x)", \\ & \quad "Price(x, y) \wedge \\ & \quad \forall (x', y', X') \\ & \quad E(x') \wedge Price(x', y') \wedge C' \longrightarrow y < y'") \end{aligned}$$

In the rule R13, we have considered that the flight duration is especially interesting for *businessman*.

$$\begin{aligned} \text{R13: } \forall (q, x, E, C, C', X') \\ & \text{entity}(q, "E(x)") \wedge \text{condition}(q, C) \wedge \text{Variant}(C, C', X') \wedge \\ & \text{User-Type}(Business-man) \\ & \longrightarrow \text{Opt-Criterium}(q, "Flight(x)", \\ & \quad "Duration(x, y) \wedge \\ & \quad \forall (x', y', X') \\ & \quad E(x') \wedge Duration(x', y') \wedge C' \longrightarrow y < y'") \end{aligned}$$

The meta predicate *User-Type* is used to define different types of users. For example, we could have the following types:

- *User-type (Tourist)* (User travels for tourism reason).
- *User-type (Businessman)* (User is a businessman)

There is also the rule R14, which is independent of a particular application domain, for expressing that an optimization criterium can be inherited by more specific entity types.

$$\begin{aligned} \text{R14: } \forall (q, e_1, e'_1, c, x) \\ & \text{Opt-Criterium}(q, "e_1(x)", c) \wedge \text{ISA}(e'_1, e_1) \\ & \longrightarrow \text{Opt-Criterium}(q, "e'_1(x)", c) \end{aligned}$$

The rules R15 and R16 express how to transform a query when an optimization criterium is used (respectively in the case of an entity type transformation and in the case of a condition transformation).

$$\begin{aligned} \text{R15: } \forall (q, e_1, e_2, C) \\ & \text{Neighbour-Ent}(q, e_1, e_2) \wedge \\ & \text{Opt-Criterium}(q, e_2, C) \\ & \longrightarrow \text{Transfl-Cond}(q, e_1, e_2, \text{True}, C) \end{aligned}$$

$$\begin{aligned}
\text{R16: } & \forall (q, e, c_1, c_2, C) \\
& \text{entity}(q, e) \wedge \\
& \text{Neighbour-Cond}(q, e, c_1, c_2) \wedge \\
& \text{Opt-Criterion}(q, e, C) \\
& \longrightarrow \text{Transf2-Cond}(q, c_1, c_2, \text{True}, C)
\end{aligned}$$

These rules say that the optimization criterium formula C has to be added to the condition part of the transformed query (by substituting the tautology formula “*True*” by the C formula).

When an optimization criterium is inserted in the transform query, it is useful to provide the user with the value of the optimized attribute; so, the user will have the possibility to compare the advantage of the neighbourhood entities with respect to the optimization criterium.

To provide this interesting comparative information, we introduce the meta predicate:

$$\text{Att-Int}(q, E(x), I)$$

where I is a comparative information useful to give for the entities of type E .

We use the rules R17 and R18, which are independent of a particular application domain, to express what is the interesting information respectively in the case of an entity type transformation and in the case of a condition transformation.

$$\begin{aligned}
\text{R17: } & \forall (q, e, c_1, c_2, C, I) \\
& \text{Neighbour-Cond}(q, e, c_1, c_2) \wedge \\
& \text{Opt-Criterion}(q, e, C) \wedge \\
& \text{Info-Int}(C, I) \\
& \longrightarrow \text{Att-Int}(q, e, I)
\end{aligned}$$

$$\begin{aligned}
\text{R18: } & \forall (q, e_1, e_2, C, I) \\
& \text{Neighbour-Ent}(q, e_1, e_2) \wedge \\
& \text{Opt-Criterion}(q, e_2, C) \wedge \\
& \text{Info-Int}(C, I) \\
& \longrightarrow \text{Att-Int}(q, e_1, I)
\end{aligned}$$

We have to notice that comparative information must be provided for transformed query answers, but also for initial query answers.

6 Conclusion

We have presented a method and a formalism to define query transformations to provide users more information than it is explicitly requested by their queries. This method allows to design a system having a more cooperative behaviour than standard Relational DBMSs.

The method is based on the idea that the interesting information strongly depends on the context, and people having a long experience in providing information can explicit their expertise in our formalism.

The selected formalism is First Order Logic, but we distinguish an object level language and a meta level language. The meta level allows to

express knowledge about the meaning of the queries. This formalism has two important advantages: it offers a very precise semantics, and it can be easily implemented using existing derivation mechanisms like Prolog. We show, in the Annex, examples obtained from a first version of a prototype implemented in this language.

If we compare this method with the work presented by A.Motro in [7], it can be noticed that, in his work, interesting information is defined using a measure on attribute values and weights for each attributes; then, computation rules and a threshold define interesting entities. This approach has the same objective of what we call “condition transformation” but it cannot take into account the expertise of an expert; except in defining the attribute weights. Moreover this method doesn’t provide entities of a neighbour entity type.

In [9], L.Siklossy presents, through examples, a similar approach but there is no formalization and no general methodology that could be exported to other application domains. One can notice that the logical representation of the rules we use allows to easily change the rules when the application domain changes, because there is a clear distinction between the rules which are application dependent and those which are not.

References

- [1] K. Bowen and R. Kowalski. Amalgamating language and metalanguage in Logic Programming. In Clark, editor, *Logic Programming*, Tärnlund, 1980.
- [2] F. Cuppens. Comment fournir des réponses coopératives aux requêtes à une base de données. Thèse de doctorat, ENSAE, 1988.
- [3] F. Cuppens. Un langage de requêtes pour obtenir des réponses intelligentes. In *Cinquièmes Journées Bases de Données Avancées*, Genève, Suisse, 1989.
- [4] F. Cuppens and R. Demolombe. Cooperative Answering: a methodology to provide intelligent access to Databases. In *Second International Conference on Expert Database Systems*, Tysons Corner, Virginia, 1988.
- [5] F. Cuppens and R. Demolombe. How to recognize topics to provide cooperative answering. *Information Systems*, 14(2), 1989.
- [6] R. Kowalski. The limitations of Logic. In J. Schmidt and C. Thanos, editors, *Proc. Workshop on Foundations of Knowledge Base Management*, Crete, 1986.
- [7] A. Motro. Supporting goal queries in Relational Databases. In *Proc. First International Conf. on Expert Database Systems*, 1986.
- [8] R. Reiter. Towards a logical reconstruction of relational database theory. In *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*. Springer Verlag, 1983.
- [9] L. Siklossy. Impertinent question-answering: justifications and theory. In *Proc. ACM National Conf.*, pages 39–44, 1978.

A Example of program execution

The query presented in this annex refers to the first example presented in the introduction. It shows an initial query, the answers to this query, and two transformed queries which provide neighbour entities by transformation of the condition part of the initial query. The answers have the form of formulas, instead of tuples. We use the syntax of predicate calculus where “&” denotes the conjunction, “Ex” denotes the existential quantifier, and “Not” denotes the negation.