

# COOPERATIVE BEHAVIOUR THROUGH REQUEST MODIFICATION

Andrea S. Hemerly<sup>1</sup>, Marco A. Casanova<sup>1</sup> and Antonio L. Furtado<sup>2,1</sup>

<sup>1</sup>Centro Científico Rio  
IBM Brasil  
Caixa Postal 4624  
20.001, Rio de Janeiro, RJ - Brasil

<sup>2</sup>Departamento de Informática  
Pontifícia Universidade Católica do RJ  
R. Marquês de S. Vicente, 225  
22.453, Rio de Janeiro, RJ - Brasil

## ABSTRACT

An algorithm to achieve cooperative behaviour through request modification is described. Here cooperation basically means to contribute to the achievement of the users' goals and plans. The algorithm is rule-driven and operates in a context far richer than conventional databases. The algorithm is fully operational and it was developed as part of a larger project that investigates knowledge based methods to create cooperative information systems.

## 1. INTRODUCTION

We describe in this paper an algorithm to achieve cooperative behaviour through request modification. An information system exhibits cooperative behaviour [AP,BJ,CCL,We] to the extent that it interacts with users in ways that:

- 1) contribute to the achievement of the users' goals and plans;
- 2) keep the users' understanding of the system in harmony with the definition and contents of the system, avoiding misconceptions and contributing to a fuller and at the same time more efficient usage;
- 3) conform to the established integrity and authorization constraints.

In what follows, we refer to a query or update command as a user *request*.

To achieve cooperative behaviour, the algorithm we propose does not execute a request literally, but rather it transforms the request appropriately, guided by a set of modification rules. The algorithm may modify a request by invoking rules before the request is

actually executed, after the request is successfully executed, or even after a failed execution.

In addition to the request itself, each rule has access to information taken from a rich *database application context* involving far more than the conventional database components. The language used to specify the various components of a database application context and to formulate requests is based on an extended entity-relationship data model and it may be considered object-oriented, according to the criteria followed by [GN].

The request modification algorithm is fully operational and it was developed as part of project NICE, whose purpose is to investigate knowledge based methods to reduce the cost of developing "help desks" and similar advanced database interfaces. The current prototype is implemented by a combination of IBM Prolog and SQL/DS, enhanced by a number of tools, the most important being a plan and schedule generation algorithm, a temporal database package, a query-the-user facility and a session-monitoring feature which, when active, keeps the session logs and allows the creation and triggering of demons.

In the present paper we concentrate on the use of rules to modify a request. References [CF,FC,TF] describe, respectively, the project NICE, the plan and schedule generator and the temporal database structures and algorithms.

The notion of request modification in database management systems was first proposed in [St] to enforce integrity constraints. Cooperative query processing is achieved in [CCL] by defining a rich conceptual model and a companion inference technique to specialize or generalize queries. A similar approach is followed in [CD], which also bases query transformation on additional information provided with the schema, under the form of common topics of interest. In another direction, the generalization of failed queries to achieve cooperative behaviour is investigated in [Mo]. A natural language database query system which recognizes users' presuppositions about the application domain is also described in [Ka]. The algorithm we describe can profit from these approaches since the inference techniques of [CCL] and the transformation rules of [CD] can be directly translated into rules fired before the execution of the request, if we incorporate the extensions they propose to our conceptual model, and the techniques in [Ka,Mo] are conveniently handled by rules invoked after a failed execution. In fact, references [CCL] and [CD] teach us that enriching the conceptual model is a very powerful strategy to organize and simplify the definition of useful domain-independent request modification rules. When compared with each of these references, we should stress that the algorithm described in this paper treats both queries and updates, and improves the cooperative behaviour by coordinating the use of rules in various stages of the execution

of a request. The treatment of updates in fact raises very interesting issues since it involves plan generation techniques to achieve users' intentions (see [FC]).

We organize the rest of this paper as follows. We briefly present in section 2 how the various components of a database application are described. We show in section 3 a simple database application together with examples of rules for request modification. Then, we outline in section 4 the request modification algorithm. We leave the conclusions to section 5.

## **2. COMPONENTS OF A DATABASE APPLICATION**

The specification of an application in our prototype system consists of a database context (a description of the conceptual schema, the external schemas, the application-oriented operations, etc...), a set of rules that capture the desired cooperative behaviour specific to the application and, optionally, a user interface written in Prolog. The prototype system offers further rules that capture cooperative behaviour independent of the applications, a stand-alone user interface and a request modification algorithm that controls the processing of requests through the rules specified.

This section outlines how to define a database application context and the types of rules the prototype handles.

### **2.1 The Context of Database Applications**

The *context* of a database application contains:

- 1) the database conceptual schema, the external schemas and the application integrity constraints;
- 2) the factual database;
- 3) pre-defined application-oriented operations;
- 4) a model of the application domain;
- 5) users' profiles, including, for each user, his often faulty understanding of his external schema;
- 6) a repertoire of patterns of typical plans, either pre-established by the system designers or recorded from past interactions with users;

and, for each user currently accessing the system:

- 7) demons to alert the user of special conditions;
- 8) a session log registering requests from and responses to the user.

The current implementation will adopt a uniform way, through the use of clauses along the lines of Prolog, to specify all components of the context. The format for the clauses describing the first three components are outlined in what follows.

The conceptual model is basically the entity-relationship model with *is-a* hierarchies and a few other minor extensions, such as *roles* for the entity classes participating in relationship class. The conceptual schema of a database is specified through clauses that define the entity and relationship classes that exist and the structure of the is-a hierarchies. An external schema may in general be defined by a Prolog procedure.

A *fact* indicates the existence of either an entity or a relationship instance, or captures that one such instance has a certain value for an indicated attribute. Each fact has an associated timestamp. A *database* for a conceptual schema is a set of facts, possibly with different timestamps.

Integrity constraints may also be directly declared, again using clauses, which indicate conjunctions of facts that are not valid and, therefore, should not be reached by any operation or sequence of operations.

In the spirit of abstract data types, the operations allowed on a database are pre-defined. Each application-oriented operation *O* is also specified by a set of clauses, which indicate the facts that are added and deleted by *O* (i.e., the effects of *O*), the pre-conditions for the execution of *O*, in terms of facts that should or should not hold and, optionally, lower and upper time bounds for the execution of *O*.

An *instance* *O'* of an operation *O* is a set of clauses obtained by uniformly substituting the variables occurring in the clauses that specify *O* by variable-free terms. We stress that the clauses that form an operation instance have no free variables.

For example, the following three clauses define an operation to enroll a student *S* into a course *C* (both *S* and *C* are variables; the syntax used below is a slight simplification of that accepted by the current implementation):

```
precond(enroll(S,C), student(S)).  
precond(enroll(S,C), course(C)).  
added(takes[student(S),course(C)],enroll(S,C)).
```

The first two clauses say that the operation is executed only if there are facts in the database indicating that *S* and *C* are indeed defined as a student and a course, whereas the last clause says that, after executing *enroll(S,C)*, the fact *takes[student(S),course(C)]* should be added to the database.

The following set of clauses then define an instance of `enroll` (both `s1` and `c3` are constants):

```
precond(enroll(s1,c3),student(s1)).
precond(enroll(s1,c3),course(c3)).
added(takes[student(s1),course(c3)],enroll(s1,c3)).
```

Finally, we remark that the current implementation does not yet support user profiles and typical plans and it covers a model of the application domain only through the definition of specific rules, as discussed in more detail in section 3. For each user, the system records the session log and the demons created during the processing of his requests, again in clause form. In the context of cooperative request processing, demons are typically used to warn a user of a special situation and are automatically created.

## 2.2 Request Modification and Template Rules

We identify two broad classes of rules: the *request modification rules* (*RM-rules*), that indicate the transformations the requests may suffer, and the *template rules*, that define how the information obtained in response to a request must be edited. Orthogonally, we distinguish between *domain-independent rules*, motivated by the database model adopted, and *domain-dependent rules*, that are specific to the application domain on hand.

The prototype system implements a host of domain-independent rules. Rather than enumerating them here, we offer the following general comments (see also section 3). The ability to define a useful set of domain-independent rules is directly dependent on the richness of the conceptual model adopted. In fact, as advocated in [CCL,CD], cooperative query processing can be achieved by defining a rich conceptual model and defining domain-independent pre-query rules that capture inference patterns the model allows. The efforts of the database application designer can then be channeled to defining the additional information that accompanies the conceptual structures. Domain-independent rules can also be defined to handle users' presuppositions, that is, the impressions about the domain the users have. A cooperative system should correct any presupposition it believes to be false, rather than simply failing. A more detailed discussion about presuppositions can be found in [Ka]. Finally, the generalization of failed queries, investigated in [Mo], is also a source of domain-independent rules.

As for the domain-dependent rules, the database designer is responsible for them. The current prototype requires some knowledge of Prolog to describe such rules, but we plan to introduce a special language to minimize the designer's efforts.

We further classify RM-rules according to the type of request to which they apply and to the execution phase where they are used. Thus, based on the first criterion, we have

*query rules* or *apply rules* and, based on the second criterion, *pre-rules*, which are invoked before the execution of a request, *s-post-rules*, that follow the successful execution of a request, and *f-post-rules*, that follow execution in case of failure.

One may use pre-rules to correct a request, or simply to translate a user-friendly external syntax into the standard syntax. One may also use pre-rules to complement a request, that is, to supply additional information, for queries and updates, and to execute additional changes, for updates only. Since the need to complement a request may be detected only after executing the request, one may alternatively define s-post-rules to generate and execute new requests.

Finally, when a request of a given class fails, one may define an f-post-rule that generates an alternative request that accomplishes, as much as possible, the same purposes. One may also create an f-post-rule that proposes to the user the creation of a demon that will alert him when the obstacles to execute the request cease to exist, if no alternatives exist or all fail, but the failure is due to removable obstacles. A removable obstacle is, for example, a pre-condition that does not hold when the request is posed, but that may perhaps hold in the future. Moreover, one may define an f-post-rule that at least proceeds to explain why the request failed, if the obstacles are known to be of a persistent nature.

To summarize, the request modification algorithm will first successively apply all possible pre-rules to the request given as input, producing a new request. Then, it will execute the modified request over the database. After a successful execution, it may execute new actions, as a result of cumulatively processing s-post rules. In case of a failed execution, it may apply f-post-rules one at a time to offer alternatives to the request. Finally, if everything fails, it may cumulatively try f-post-rules to compensate the failure. The algorithm will also use template rules to edit messages generated by RM-rules.

### **3. SAMPLE REQUESTS AND RESPONSES**

To illustrate requests and rules, and to show how the rules interact in a cooperative way, we specify in this section a simple-minded academic database and describe several requests, together with the transformations determined by rules. We prefer to give a verbal description, instead of a full definition in the formal language implemented, to avoid excessive detail. All rules described in the examples, among others, are operational in the experimental prototype. The formal specification of requests, rules and the request modification algorithm itself is presented in section 4.

### 3.1. Example of a Database Application

Consider the academic database of a university department whose relevant facts describe students with their ids and names, courses with their number of credits, which students take which courses with which mark, and laboratory courses and their admittance requirements. Laboratory courses are treated as a specialization of courses and, hence, they inherit the number of credits as attribute.

The examples will refer to the following operations: *offer* to create a course; *enroll* to enroll a student in a course; and *transfer* to transfer a student from a course to another. The pre-condition to the last two operations is that the student exists and the course the student will be taking must be offered. In addition, the execution strategy for operation instances (see section 4.1) rejects enrolling a student in the same course more than once and transferring a student from a course to the same course.

The examples also depend on a constraint establishing that no student can take both **C1** and **C2** at the same time. Finally, the users of the application are the department chairman, teachers and students.

Now suppose the database contains the following facts:

- **S1** is a student;
- courses **C1** and **C2** are offered with 2 credits;
- student **S1** takes course **C2**.

### 3.2. Examples of the use of RM- and Template rules

This section exemplifies how the request modification algorithm (or, briefly, **RMA**) uses rules to modify requests, at different stages of the processing of a request. Each example includes a query or update (indicated by **Q** and **U**, respectively), followed by a description and a classification of the rules applied. The classification of each rule indicates when it is applied and how it depends on the domain.

The first example involves pre and f-post rules. The pre-query rule corrects a request based solely on schema information. The f-post-query rule corrects a type of false presupposition.

### Example 1:

**Q1** - "Does course C4 have admittance requirement AC4?"

The RMA first applies a pre-query rule correcting the query to "Does lab C4 has admittance requirement AC4?". This rule recognizes the rather common error of referring to a more general entity class when an attribute that belongs to a more specialized class is involved. The modified query will fail for the sample database. Note that the user has assumed in this case that lab C4 is offered, which is not true in the sample database. The RMA then applies an f-post-query rule, recognizing the false presupposition of the user about the existence of C4, and replies "There is no lab C4", which is more cooperative than just returning FAIL.

The rule used to check the schema information contained in the query is:

**rule 1** - a domain-independent pre-query rule

If a query wrongly references an attribute A as belonging to an entity class E, where in fact A belongs to an entity class F such that F is-a E, the query is corrected by replacing E by F and a message reminding the user about the existence of this "is-a" link is issued. If such correction is not possible, a message is issued to tell the user which error has occurred: entity class E does not exist; attribute A belongs to some other entity class E'; or attribute A does not exist.

The repertoire of the prototype also includes a similar rule for relationship classes and a second rule, invoked when "apply" is executed, to check if the operation referred to indeed exists and if the operation parameters are correct.

The rule used to correct the type of false presupposition above is:

**rule 2** - a domain-independent f-post-query rule

If an entity instance is not present, even in a relationship, then a message is issued relating that.

In fact, the current implementation includes rules that explain why a query involving a relationship fails. They give priority to testing if the query fails because one of the entities mentioned is not present in the database, and then to testing if the entities exist, but are not related.

The following example uses two types of f-post rules: one to offer an alternative to a failed request and another to compensate its failure. The rules involve pre-conditions

and integrity constraints. Besides guaranteeing the integrity and authorization constraints, as expected, the system processes an update in a cooperative way, giving explanations, monitoring triggers or offering alternative operations. In the latter case, the system takes advantage of the plan generator, described in [FC].

**Example 2:**

**U1** - "Enroll S1 in C1."

Since no pre-rule is applicable, the RMA tries to execute the update in its original form. However, this update request fails in view of the integrity constraint forbidding students to simultaneously take both C1 and C2. Thus, the RMA applies an f-post-apply rule, where an alternative found by the plan generator is to transfer S1 from C2 to C1. The user is asked to confirm if he agrees with the execution of the alternative. Suppose that the user disagrees with it causing this step to fail. Then, the RMA applies an f-post rule explaining why the update fails.

The rules applied in this case are:

**rule 3** - a domain-independent f-post-apply rule

If the application of an operation instance 0 fails, the plan generator tries to find an alternative operation instance that achieves at least all the effects of 0.

**rule 4** - a domain-independent f-post-apply rule

If an update fails because the resulting state would violate an integrity constraint, a message is issued to explain that this is the reason for the failure. But if the update fails because a pre-condition is not yet fulfilled, a demon may be created, that will be awakened when the pre-condition becomes true to warn the user when he can try the update again; the demon will then be destroyed.

Another relevant strategy to achieve cooperative behavior is to consider the focus of the requests. In the next example, the algorithm uses a f-post-query rule which tries to capture the focus of the user requests during a session.

### Example 3:

**Q2** - "Is course C5 being offered with 2 credits?"

Again, no pre-rule is applicable and the RMA tries to execute the query in its original form. This query fails for the sample database. However, suppose that the session's log registers that the user has already submitted another query asking if some other course has 2 credits. If this is the case, the user is probably trying to locate some course with 2 credits. Thus, the RMA applies an f-post-apply rule, which generates an alternative query "Find some course with 2 credits". The new query succeeds and C1 is returned as a course with 2 credits.

The rule applied in this case is:

**rule 5** - a domain-independent f-post-query rule

If a query Q asking whether a given entity instance has a given value for an attribute fails, and the session log indicates that a similar query Q' had been asked before about a different entity instance and that Q' also failed, an alternative query Q'' is created by generalizing Q to ask for *some* entity instance meeting the requirement. The most specific generalization algorithm [WM] is used to select which element of Q can be transformed into a variable.

The domain-dependent rules may or may not be specific to a certain class of users. The pre-query rule applied in the next example is specific to the students of our simple academic database. The example also shows a template rule.

### Example 4:

**Q3** - "Are courses C1 and C2 being offered?"

When a student asks about a course, he is probably considering to take it and, hence, the number of credits assigned is relevant to his decision. Thus, the RMA applies a pre-query rule which complements the query to "Are courses C1 and C2 being offered? with how many credits?". The modified query succeeds for the sample database with unmodified answer "Course C1 is offered with 2 credits and course C2 is offered with 2 credits". However, the RMA uses a template rule which edits the answer and change it to "all with credits = 2", since both courses have 2 credits. The answer becomes more informative by stressing that the number of credits is the same.

The rules used are:

**rule 6** - a domain-dependent pre-query rule

If a student asks if a course is being offered, the query is complemented to also return the number of credits.

**rule 7** - a domain-independent template rule

If a query asks for the values of the same attributes for several entity instances, and it succeeds with the value of one or more attributes being the same for all entity instances, then the answer is edited to express concisely that such values coincide. The most specific generalization algorithm is used for this purpose.

Another example of a domain-independent template rule is one that uses the roles of the entities in a relationship to edit an answer in a suitable English form.

#### 4. THE REQUEST MODIFICATION ALGORITHM

We describe in this section the algorithm to modify and execute a request. To facilitate the discussion, we first describe how the algorithm processes a single request expression, then how it applies a single rule to a request expression and finally what strategies it follows to process several rules against one request expression. Then, we separately outline the major steps of the algorithm and the backtracking strategies the algorithm adopts.

##### 4.1 Execution of a single request

We first formally introduce the syntax of requests and then outline how they are processed, ignoring modifications for the moment.

A *query expression* is a Prolog expression involving one or more database facts.

Now, a *request execution command*, or simply a *request*, is an expression of the form:

<request type>( <expression>, <information> )

where, in the current implementation:

<request type> is either query or apply. Requests of the first type are naturally called *queries*, whereas those of the second type are called *updates*;  
<expression> is a query expression, if the type is query,  
or an operation instance, if the type is apply;

`<information>` is a Prolog expression.

The first parameter is called the *request expression* and the second parameter is called the *information parameter* of the request.

Users may either directly submit requests through the stand-alone interface the prototype offers, or interact with the database through a special interface. In the second case, the interface would be defined in Prolog and the requests included in Prolog clauses as any other goal. We should also add that most often the information parameter of a request will simply be a variable, whose value will be displayed after successful execution. It will typically be a more complex expression, perhaps containing free variables, if the request is used in a Prolog clause, that may include other terms where some of these variables also occur.

Given a Prolog expression  $E$  and a substitution  $\beta$ , we will denote by  $E\beta$  the expression obtained by applying  $\beta$  to  $E$ .

Ignoring modifications, a query expression  $Q$ , specified in a query, is executed as a Prolog goal against the database (considered as a Prolog program). This roughly means searching for a substitution  $\beta$  for the free variables in  $Q$  such that  $Q\beta$  is a logical consequence of the database. If the search succeeds,  $Q\beta$  will be returned ( $Q\beta$  is then called a *solution* of the query expression). Otherwise, FAIL will be returned. The search may also be repeated to find another substitution, if one exists at all, until no new substitution can be found. In the special case that  $Q$  has no free variables, the strategy is simply to test if  $Q$  is a logical consequence of the database. If so,  $Q$  will be returned ( $Q$  is also called a *solution*). Otherwise, FAIL will be returned as before.

The strategy to process an operation instance  $O$ , given in an update, follows [VF]. If the pre-conditions of  $O$  do not hold for the current database, or some of the effects of  $O$  already hold in the current database, or the effects of  $O$  would violate some integrity constraint, no change is produced and FAIL is returned. Otherwise, facts are added and deleted, as specified by the clauses of  $O$ , and all other facts are left unchanged (this solves the so-called frame problem [Ko] as expected). For compatibility with the processing of query expressions,  $O$  will be returned ( $O$  is also called a *solution*).

#### **4.2 Processing a single RM-rule against a request expression**

We first introduce the formal syntax for request modification rules and explain how to apply a rule to a request. We then repeat the discussion for template rules.

The format of a request modification (RM-) rule is:

`<rule-type>(<in-pattern>,<out-pattern>,<action>,<message>) <- <condition>`

where, in the current implementation:

- `<rule-type>` is the *type* of the rule, taken from the set {pre\_query, s\_post\_query, f\_post\_query, pre\_apply, s\_post\_apply, f\_post\_apply};
- `<in-pattern>` is the *input pattern* of the rule, a Prolog term that identifies a class of request expressions;
- `<out-pattern>` is the *output pattern* of the rule, a Prolog term that will result in the modified request expression;
- `<action>` is the *action* of the rule, a Prolog procedure that, acting on `<out-pattern>` and `<message>`, will create the modified request expression and a message;
- `<message>` is the *message* of the rule, a Prolog term that will evaluate to a character string;
- `<condition>` is the *condition* of the rule, an optional Prolog expression that should be satisfied to apply the rule.

We note that the output pattern and the action may have the value "true", the message may consist of the null string and the condition may be absent. In any of these cases, we say that the component is *trivial*.

Now let  $r$  be an RM-rule with the above format and let  $E$  be a query expression or an operation instance. The application of  $r$  to  $E$  produces a new request expression  $E''$  along the following steps:

- 1) *Find if the rule is applicable to the request expression* - First try to unify  $E$  with `<in-pattern>` and test if `<condition>` holds. If both succeed, the rule is applicable to the request expression. Let  $A'$ ,  $E'$  and  $M'$  be the expressions obtained by applying the resulting variable substitutions to `<action>`, `<out-pattern>` and `<message>`.
- 2) *Execute the action* - If  $A'$  is not trivial, execute it and let  $E''$  and  $M''$  be the expressions resulting by applying the variable substitutions thus obtained to  $E'$  and  $M'$ . Otherwise, let these expressions be  $E'$  and  $M'$  themselves.
- 3) *Generate the output* - Output the new request expression  $E''$  and, if  $M''$  is not trivial, display it, perhaps edited by a template rule (as discussed below).

The format of a template rule in turn is:

`template(<information>,<display>) <- <condition>`

where, in the current implementation:

- `<information>` is the *information* parameter of the rule, a Prolog expression;

- <display> is the *display* of the rule, a Prolog expression that will evaluate to the character string to be displayed and that typically translates the information parameter into a natural language sentence;
- <condition> is the *condition* of the rule, an optional Prolog expression that should be satisfied to apply the rule.

Template rules are invoked either manually using the `pwrite` command, whose only argument is a Prolog expression, or automatically during the processing of RM-rules. The execution of `pwrite(A)` proceeds as follows:

- 1) search for a template rule whose information parameter unifies with *A* and whose condition, if present, succeeds;
- 2) if one exists, exhibit the display of the rule, after applying all substitutions obtained from the previous step;
- 3) otherwise display the argument *A* in its original form.

The invocation of a template rule during the processing of an RM-rule follows likewise.

### 4.3 Strategies to process several rules against a request expression

Given a request expression *E*, there are three different strategies to process a set *S* of RM-rules against *E*:

- *successive* - the request expression *E* is transformed into a final request expression  $E_n$  along a sequence of transformations  $E, E_1, E_2, \dots, E_n$ . Each transformation is obtained by applying one rule. The same rule may be applicable several times. The process terminates when some request expression is obtained to which no rule in *S* is applicable.
- *cumulative* - all rules in *S* are tried against *E* and the result is the set  $\{E_1, E_2, \dots, E_n\}$  containing all requests expressions  $E_i$  such that *E* can be successfully transformed into  $E_i$  by applying one rule.
- *alternative* - each rule in *S* is tried, in turn, against *E*. At each time, only one successful transformation is considered.

### 4.4 The core algorithm

The request modification algorithm is basically the same for query and update requests. The difference, which is the object of the next section, is how backtracking is handled in each case.

Let *R* be a request with request expression *E* and information parameter *I*. The algorithm processes *R* like a Prolog goal, that is, it either returns FAIL, or returns *R*, with *I*

replaced by some term via unification (E is not affected, even if it contains free variables). The main steps are:

1) *Correct and complement the original request:*

Process pre-rules against the request expression E, using the successive processing strategy, to obtain the modified request expression E' (which remains E itself, if no pre-rule applies to E). The invocation of a pre-rule may determine that the request should fail at this point.

2) *Execute the (possibly) modified request expression:*

- a) Execute the modified request expression E', as explained in a previous subsection.
- b) If the execution succeeds, returning a solution E'' for E', then proceed to step 3.
- c) Otherwise, proceed to step 4.

3) *Complement the response:*

- a) Apply s-post-rules to E'', using the cumulative processing strategy. Each application of an s-post-rule to E'' that succeeds returns a new request expression. Let  $E_1, E_2, \dots, E_m$  be the sequence of request expressions thus obtained.
- b) Execute each  $E_i$ , as explained in a previous subsection, returning a solution  $E''_i$  for  $E_i$  or FAIL. Let  $E''_{j_1}, E''_{j_2}, \dots, E''_{j_n}$  be the sequence of solutions thus obtained (that is, ignore the request expressions  $E_i$  that failed).
- c) Unify the information parameter I of the original request R with  $E'' \& E''_{j_1} \& E''_{j_2} \& \dots \& E''_{j_n}$  or simply with E'', if the sequence is empty.
- d) If the unification fails, proceed to step 4.
- e) Otherwise stop and return the original request R, with I modified by the above unification.

4) *Try alternative:*

- a) Apply f-post-rules with a non-trivial output pattern to E', using the alternative processing strategy.
- b) Execute each request expression thus obtained, without invoking the request modification process recursively, until one succeeds and returns a solution E'' that unifies with the information parameter I of the original request R, or all have been tried. For updates, each alternative is not automatically executed, but rather shown to the user, who must reply whether or not he wants its execution.
- c) If the previous step fails, proceed to step 5.
- d) Otherwise stop and return the original request R, with I unified with E''.

5) *Compensate failure:*

- a) apply f-post-rules with a trivial output pattern to E', using the cumulative processing strategy. Such rules may display messages, create demons or produce other side-effects.
- b) Stop and return FAIL.

## 4.5 The backtracking discipline

The algorithm uses distinct backtracking strategies for queries and updates. The strategy is rather liberal for queries, since they leave no effect on the stored data and are often submitted with the mere intent of browsing. Therefore, we begin by discussing how backtracking works for the various steps of the algorithm with respect to queries.

The correct/complement step cannot be forced to backtrack since a single sequence of transformations is assumed. This decision simplifies the verification that the transformation process terminates and avoids the combinatorial explosion that would otherwise arise if we considered all possible sequences of transformations. On the other hand, it does not ensure that the modified query expression obtained is the best possible in all cases.

The execute step may be forced to backtrack, if there is more than one solution for the (modified) query expression. For each solution, the complement response step (which does not by itself backtrack, since its cumulative strategy does not require that) is of course repeated.

The try alternative step may also be forced to backtrack, so that each alternative can be seen. To provide maximum freedom, we established in addition that the try alternative step may be reached, if backtracking is provoked beyond the last successful invocation of the execute step. Moreover, the compensate failure step may also be reached, if backtracking is provoked beyond the last successful invocation of the try alternative step.

For updates, the essential point is that only one successful execution of an operation instance can take place when the "apply" command is invoked. So, if the execute step succeeds for the (modified) operation instance, the complement response step takes place and then the algorithm stops. Otherwise the try alternative step is entered. Moreover, if one alternative is accepted, the algorithm executes it and stops. If no alternatives exist or if they are all rejected, the compensate failure step is performed once.

## 6. CONCLUSION

A first version of the NICE prototype, whose core is the request modification algorithm described in this paper, is operational.

Our efforts in the immediate future will address the following points. We first intend to extend the repertoire of the domain-independent RM-rules since they have a very broad impact. We are also developing a user modelling theory and a method to design and

implement user profiles. We must also learn how to further customize cooperative behaviour to user classes and individual users [KW,NS]. Finally, we will also devote some effort to capture the idea of typical plans.

As the research proceeds, we expect to consider more complex examples that explore more deeply the many interesting aspects of cooperative behaviour that arise when time is considered, and which are tractable through our temporal database machinery. Also, we have not yet looked at query requests involving the frame construct and frame operations supported by our query language.

## REFERENCES

- [AP] J. F. Allen and C. R. Perrault - "Analyzing intentions in utterances" - Artificial Intelligence 15, 3 (1980) 143-178.
- [BJ] L. Bolc and M. Jarke (eds.) - "Cooperative Interfaces to Information Systems" - Springer-Verlag (1986).
- [CCL] W. Chu, Q. Chen and R-C. Lee, "Cooperative Query Answering via Type Abstraction Hierarchy", Proc. Int. Working Conference on Cooperating Knowledge based Systems, Univ. Keele, UK (1990).
- [CF] M. A. Casanova and A. L. Furtado - "An Information System Environment based on Plan Generation", Proc. Int. Working Conference on Cooperating Knowledge based Systems, Keele, UK (1990).
- [CD] F. Cuppens and R. Demolombe - "Cooperative answering: a methodology to provide intelligent access to databases" - Proc. of the Second International Conference on Expert Database Systems - L. Kerschberg (ed.) - Benjamin/Cummings (1989) 621-643.
- [FC] A. L. Furtado and M. A. Casanova - "Plan and schedule generation over temporal databases" - Proc. of the 9th International Conference on Entity-Relationship Approach (1990).
- [GN] H. Gallaire and J. M. Nicolas - "How to look at deductive databases" - in "Foundations of Knowledge Base Management" - J. W. Schmidt and C. Thanos (eds.) - Springer-Verlag (1989) 119-130.
- [Ka] S. J. Kaplan - "Cooperative Responses from a Portable Natural Language Query System" - Artificial Intelligence 19, 2 (1982) 165-187.
- [Ko] R. Kowalski - "Logic for problem-solving" - North-Holland Pub. Co. (1979).
- [KW] A. Kobsa and W. Wahlster (eds.) - "User Models in Dialog Systems" - Springer-Verlag (1989).
- [Mo] A. Motro - "Query generalization: a technique for handling query failure" - Proc. First International Workshop on Expert Database Systems (1984) 314-325.

- [NS] E. J. Neuhold and M. Schrefl - "Dynamic derivation of personalized views" - Proc. of the 14th VLDB Conference (1988) 183-194.
- [St] M. R. Stonebraker - "Implementation of integrity by query modification" - Proc. ACM SIGMOD International Conference on Management of Data (1975).
- [TF] L. Tucheran and A. L. Furtado - "Update-oriented database structures" - Proc. of the Second International Conference on Expert Database Systems (1988) 185-203.
- [VF] P. A. S. Veloso and A. L. Furtado - "Towards simpler and yet complete formal specifications" - in "Information systems: theoretical and formal aspects" - A. Sernadas, J. Bubenko and A. Olive (eds.) - North-Holland Pub. Co. (1985) 175-189.
- [We] B. L. Webber - "Questions, answers and responses: interacting with knowledge base systems" - in "On knowledge base management systems" - M. L. Brodie and J. Mylopoulos (eds.) - Springer (1986).
- [WM] A. Walker, M. McCord, J. F. Sowa and W. G. Wilson - "Knowledge systems and Prolog" - Addison-Wesley Pub. Co. (1987).