CAiSE*94, 6–10 June 1994, Utrecht, The Netherlands.

Designing a User-Oriented Query Modification Facility in Object-Oriented Database Systems

K. Aberer, W. Klas, A. L. Furtado[1]

GMD-IPSI, Dolivostr. 15, 64293 Darmstadt, Germany,
E-Mail: {aberer, klas, furtado}@darmstadt.gmd.de

Abstract. The introduction of user-assisting features into database systems is discussed along two stages. The first stage involves a basic facility that can be used with standard database systems, whereas in the second stage such features are expanded in order to cope with object-oriented systems, adopting semantically richer data models. Examples involving categorization and role-specialization semantic hierarchies illustrate the discussion. A class/metaclass architecture, such as that of the VODAK database system, an algebraic view of query processing and an extension of the object-oriented data model by rule systems are shown to be particularly suitable to design and implement user assistance on the database schema level.

1 Introduction

As increasingly complex databases are designed, end users may find them too difficult to use. One runs the risk to produce systems that are rich in terms of the information they contain, but that no one is able to use appropriately. The problems of accessing data in a database system are manifold: The user may not have the right conceptual understanding of what is in the database, he may not know or may not be able to use the right terminology to exactly name the constituents of a database schema (e.g., property names, class and type names), or he even may not know exactly how to express his information needs in terms of the query language offered by the database system. These problems are also well known in the framework of information retrieval and several approaches have been proposed to overcome these difficulties. An obvious solution is to interpose between user and system a module that acts as an ‘‘assistant”. Such an assistant can help the user in formulating requests by means of a query statement which is then submitted to the database system, or it can modify the original query by means of abstractions or refinements, thus submitting queries which result in more information than the original query would have provided. In general, one can identify the following kinds of approaches for this problem: (1) *Interactively* support the user in formulating the query so that the submitted query be the best possible approximation to the user’s information need. Then, process the query using conventional query processing techniques and return the exact result for the formulated query. The presentation of the result may be enhanced in order to make the results more understandable to the user. The whole process can be iterated in a session until the user is satisfied. (2) Take a query formulated by the user in terms of a given query language and modify the query before processing it. The modification of the query may be based on two phases: an abstraction which leads to a

[1]) Visiting from the Departamento de Informatica of the Pontificia Universidade Catolica do Rio de Janeiro.

relaxed query, i.e., it will result in a broader answer than the original query; a refinement which leads to a set of more concrete queries, whose answer is better focused on the user's interests.

In [12] the problem of facilitating the access to a database system is approached by supporting the formulation of queries using fuzzy and associative knowledge about the terminology used to set up a database schema. The goal is to construct an assistant which allows to explore a database schema and to get suggestions for formulating exact queries as required by an underlying database system. This approach corresponds to the solutions of class (1). [2] presents an interesting approach for incorporating neighborhood information and associative relationships between objects to answer user requests. The approach – which is of type (2) – is based on the idea that one can use abstractions (type/class hierarchies) from the original data to broaden a request by transformations of the original query such that the relaxed query is posed against the abstractions instead of the original data. A specific operator called *nearer* subsequently allows to narrow the relaxed answer. A major drawback of this approach is that a lot of additional information is needed to build the abstractions as well as the knowledge base containing the additional information about associations. The abstractions are hard-wired into the system, and no user-oriented query processing is supported due to the lack of a user model.

In this work, where the type (2) approach is also taken, we first demonstrate how to introduce a basic user-assisting facility, which is compatible with standard databases. We then show how to gradually adapt and expand it, to achieve a more advanced assistant module, able to work on – and take full advantage of – systems providing higher level semantics and, in particular, following object orientation. Our description of the basic facility uses the experience gained with a prototype, developed as part of a larger project (NICE) [8] that investigates knowledge-based methods to create cooperative information systems. The prototype was implemented on top of an SQL relational database, using PROLOG for the rule-based algorithms. The progressive upgrading of the basic facility towards the assistant module is part of the database projects at GMD-IPSI. Prominent among the features of the VML data model [10], which underlies the VODAK database system developed at GMD-IPSI, are the notions of classes and metaclasses, and the provision made for the specification of rules for equivalence of VML-expressions. With the help of metaclasses, it is possible to represent to a large degree the general knowledge built into semantic hierarchies, such as specialization/generalization [2,13]. With the help of equivalences, it is possible to perform rule-based transformations of database queries. Moreover, classes and metaclasses have been implemented in VODAK in a way that promotes extensibility and modularity, the latter feature being indispensable for efficiently structuring large sets of rules. In spite of the fact that our project refers to a specific system, the guidelines specified are largely applicable to object-oriented database systems supporting similar features.

The text is organized as follows. Section 2 summarizes our basic approach to user-assisting query interfaces. Section 3 is a short account of properties of specialization / generalization, as e.g. supported by the VODAK data model, and illustrates the basic idea how knowledge about specialization / generalization relationships can be exploited for user assistance. Section 4 briefly introduces the basic concepts of the VODAK model language as far as needed in our discussion. Section 5 gives the approach to design and implementation of the advanced assistant module for query modification in VML. Conclusions and directions for further work are the objective of section 6.

2 Review of Basic User-Assisting Query Interfaces

Since, in general, users do not pose questions to a database system out of idle curiosity, the first objective of a user-assisting query interface should be to identify the goals and plans of the current user. A complementary objective is to keep his understanding of the system in harmony with the intentions of the designer, so that he may fully and efficiently have access to the existing information. In the following we give a short account the basic approach for a *user-oriented context-sensitive* query processing.

2.1 Forms of Assistance

To assist the user with respect to a query, the system may offer, among others, the following services [16]:

- correct the query
- complement the query, usually expanding but sometimes restricting it to a more useful scope
- provide alternatives
- offer to monitor the database and warn the user when a specified state is reached
- explain the reasons why some situation does not hold
- undo possibly wrong assumptions of the user (misconceptions)
- prevent wrong conclusions that the user may draw from an answer (misconstruals)
- make the answers more understandable

A constant danger is the possibility to become "over-cooperative". It is imperative to keep the focus on what is indeed relevant to the user's purposes. That means, that a specific user may want to get a specific kind of assistance in a situation, while another user may not want to get the same assistance in the same situation. This obviously calls for *user-oriented* assistance in contrast to general assistance which does not consider the individual needs of a user.

2.2 Rule-driven Query Modification

A usual strategy [3], which we also adopt, is to perform query modification, a device introduced with the INGRES project [15]. Query modification involves expression manipulation for which a rule-driven approach offers itself, as followed e.g. in [8]. Rules may come from different sources; they can be:general, application dependent or user dependent. Rules may be applicable at three different phases:

pre-rules: before the query is processed, in order to correct and complement the original query.

succ-post-rules: after successful processing, in order to complement the answer through additional queries.

fail-post-rules: after failed processing, in order to try alternatives; and if failure persists, to explain.

Example 1. Let us illustrate this approach by an example as reported in [8]. The format of the rules for query modification is there given in a PROLOG-like notation:

<rule_type> (<in_pattern>, <out_pattern>, <action>) ← <condition>

where the components are as stated below:

<rule_type>	one of pre-rule, succ-post-rule, fail-post-rule,
<in_pattern>	to match the original query, if the rule is applicable,

<out_pattern>	from which the new query is built,
<action>	procedure to finish the building process,
<condition>	finishes testing applicability.

To apply one rule to the original query, the algorithm first determines if the rule currently being considered is applicable. In this case it then proceeds to create the new query. To determine if the rule is applicable, it performs two steps:

(1) it checks whether the <in_pattern> matches the original query; the pattern-matching process is done through unification as defined in logic programming;

(2) it executes <condition>, which is a logical expression whose terms may check context-sensitive conditions. The characterization of context will be given later.

After these two steps, the <out_pattern> may already have been converted into the new query, although in general it will be necessary to

(3) execute <action>, which is a procedure able to access the context. ■

2.3 Exploiting the Context of the Query

What we call ‘‘context” consists of conventional and non-conventional database components. Typical conventional components we consider the *factual database* and *data dictionary* information, like the conceptual schema, the user’s external schema, and authorization and integrity constraints. The non-conventional components can include models of the application and the user, logs of user sessions, or typical users’ plans.

Example 2. Let us illustrate the context-free pattern-matching process and the context-sensitive execution of rules according to the query modification approach taken in Example 1. We consider a failed query to find out if there are places on a flight from city amsterdam to city berlin by carrier klm on may.5th. Assume the existence of a fail-post-rule to ask about alternative flights. Let the original query and the two patterns in the rule be, respectively:

original query:	fly(amsterdam,berlin,klm,may.5th) ?
<in_pattern>:	fly(X,Y,Z,T)
<out_pattern>:	fly(X,Y,W,V)

Unifying the original query with <in_pattern> binds the variables of the latter as:

X = amsterdam, Y = berlin, Z = klm, T = may.5th.

Note that consistent substitution requires that X and Y in <out_pattern> be bound to the same values. Furthermore no values were assigned to W and V, which therefore remain free variables. Now assume that <condition> can decide from the context whether a different flight would be admissible in view of the user’s goals; in case this is true, the rule is indeed applicable (otherwise it would be dropped at this point). Finally, assume that <action> can find from the context whether the ticket for the original flight is endorsable, in which case it will unify V with T, so that the new query will be:

fly(amsterdam,berlin,W,may.5th) ?

Otherwise it will preserve the carrier originally indicated (by unifying W with Z) and try another date:

fly(amsterdam,berlin,klm,V) ? ■

Algorithms realizing rule-driven query-modification are completely generic, since they do not specify the transformations. This task is deferred to the rules. One may start with just a few rules, accessing a context with few non-conventional components. For example, we feel that at least some primitive form of session log should be maintained,

as a single query is not a convenient unit for cooperativeness/assistance [14]. As the context is enriched, new rules may be added to take advantage of whatever additional knowledge is available. Rules can rely on tools extracting or utilizing knowledge about the context, e.g.:

- a query-the-user module, whereby the system can directly learn about users' preferences [7];
- a plan-generation algorithm, able to construct plans as sequences of instances of operations which, as noted, are specified in a STRIPS-like style [4];
- an algorithm to extend unification to frames and to perform most specific generalization over terms and frames;
- a session manager to keep the log, and support an active environment through the creation of demons to perform monitoring tasks.

A prototype of a cooperative/assisting system, designed as outlined in this section, is fully operational [8]. An extension to the methodology is being studied for geographic databases [9]. Experience with the prototype there has demonstrated the usefulness of the basic facility.

However, more work was needed to cope with the additional problems – and opportunities – for user assistance arising from object-oriented systems based on semantically richer data models. With this purpose in mind, we chose to examine, as a significant benchmark, queries on database structures involving categorization and role-specialization semantic hierarchies.

3 Exploiting Specialization/Generalization Relationships for User Assistance

In the area of Semantic Data Modeling semantic hierarchies have long ago been borrowed from the area of Artificial Intelligence and adapted to the needs of databases [6]. In this section we briefly recall a number of basic features that have been identified with respect to specialization / generalization hierarchies and outline how these features are exploited for user assistance by query modification.

3.1 Basic constraints

A conventional way to model the application domain in a database is to start with a number of general classes of objects, to which base types are attached. If specialized classes are desired, they are derived from other classes by means of e.g. the ISA-relationship. For two classes in a ISA-relationship, their extensions are in a subset-relationship and their types are in a subtype-relationship. In [6] constraints are identified that ensure consistency of ISA relationships on classes, in particular it follows that they are hierarchical structures.

In addition to the basic constraints, other constraints may be imposed optionally, thus characterizing different kinds of ISA-hierarchies, e.g.:

(1) be pairwise disjoint
(2) together cover the extension of the general class
(3) have a criterion for membership
(4) form trees.

The first two constraints – disjointness and covering - are orthogonal to each other. Together they characterize a partition of the general class, in the sense that each general instance must correspond to an instance in exactly one specialized class. Constraint (3)

means that there is a procedure to determine whether an object can be a member of a given specialized class. This procedure may either depend exclusively on intrinsic characteristics of the object, or it may depend on extrinsic criteria based on how the object is related with other objects. Constraint (4) expresses that ISA edges can be required to impose a strict hierarchy, so that the classes are structured as multi-level trees. Dropping this constraint results in a partial order structure, where a class is allowed to have more than one ‘‘parent”. Constraints of the kind introduced have to be maintained by the database management system when updates are performed.

3.2 Inheritance

Another important consideration for ISA hierarchies is the inheritance of properties. In fact methods are also inherited and, for our present purposes, it is useful to note that certain methods really amount to ‘‘virtual” properties, whose values are computed by methods rather than stored in the database.

Informally speaking, the prevailing principle is that every property that is common to all specialized classes should be ‘‘factored out”, i.e. moved up to the general class. Then the term ‘‘inheritance” means that, when a query asks about such properties when referring to specialized instances, the appropriate values will be found at the level of the general instance, and duly passed down. Ambiguities may arise with non strictly hierarchical ISA, when a class may specialize more than one general class having the same property. In this case a criterion must be fixed which property value should be chosen.

A situation that is, in a sense, the inverse of inheritance may occur for some properties. The value of a property of a general instance may be synthesized from properties of one or more (if overlapping is permitted) of its specialized instances. The property Revenue of Person, for example, would be calculated as the sum of salary as Employee, plus gain as Stockholder, plus other incomes that a person may receive.

3.3 Using ISA for User-Oriented Assistance – Examples

In many cases it may happen that users are not aware of all details in many possible overlapping ISA-hierarchies. In these cases the constraints on ISA-hierarchies discussed above may readily be exploited in order to properly modify the users’ query. We give a number of examples which illustrate this fact.

Example 3. The examples below are expressed in natural language for readability. In each case we indicate the feature exploited (existence, etc.), the classes involved, the original query and the answer.

(1) *feature:* existence in another specialized class;
classes: Enterprise ISA Institution, Government Agency ISA Institution.
query: What is the address of enterprise Alpha?
answer: the address of government agency Alpha is Karlstr, Darmstadt.

(2) *feature:* location of property in specialized role;
classes: Employee ISA Person.
query: What is the salary of person John?
answer: as an employee, John earns 100.

(3) *feature:* covering;
classes: Emp-Level-1, ... , Emp-Level-n ISA Employee..
query: How many employees do not belong to any level?
answer: none; every employee is in a level group.

(4) *feature:* disjointness;
classes: same as above.
query: How many employess are in levels 4 or 5?
answer: 15; 10 in level 4 and 5 in level 5.
(5) *feature:* overlapping;
classes: Employee ISA Person, Shareholder ISA Person.
query: How many employees and how many shareholders are there?
answer: 100 employees and 15 shareholders; 5 persons are both. ■

From the above examples it is clear that there is an potential for user-assistance with regard to semantic relationships like ISA-hierarchies. In the following we will give a detailed approach for a concrete object-oriented database management system, that supports mechanisms to define such semantic relationships in a generic manner, and then show how to adapt the system for the support of user-assistance by rule-based query modification.

4 The Extendible VODAK Database Model

In order to discuss the realization of user-oriented query modification we first summarize in this section the essential features of the VODAK database system. VODAK is an object-oriented database management system which has been implemented at GMD-IPSI [10].

4.1 The VODAK Metaclass System

In VODAK specialization/generalization relationships are realized through metaclasses. We shortly introduce the concepts of classes and metaclasses as used in VODAK [11].

Classes determine the structure and behavior of their instances. More precisely, an application class determines the *application-specific* structure and behavior of its *instances*, which represent the ‘‘real world” objects dealt with within an application program, by specifying its *instance type*; it also determines the *application-specific* structure and behavior of the application class itself, for example specific object creation and initialization methods, by specifying its *own type*.

Metaclasses are used to describe the common structure and behavior of classes and their instances which may not be known at the time a metaclass is defined. Metaclasses and application classes are treated uniformly as classes. In addition, it is possible to specify the *common* structure and behavior for the instances of the instances, the so-called *metainstances*, of a metaclass by associating an object type as *instance-instance type* to the metaclass.

The *object types* that determine the instance, instance-instance and own type of classes and metaclasses can either be defined directly within the class definition, in which case we have an *in-line type definition*, or in a separate *object type declaration.* The provision of separate object type declarations in VML is also referred to as the *dual model*, which allows a clean separation of syntactic and semantic concepts in object-oriented data base schemas. For example, the same object type declaration may be reused in different (semantically unrelated) classes.

The *interface* defined for an object is the set of methods which are defined for the object, i.e., which can be executed directly for the object. It consists of the methods specified with the *own type* of the object, if this object represents a class, and the methods

specified with the *instance type* of the object's class, and the methods specified with the *instance-instance type* of the metaclass of the object's class.

In case the method is not contained in the interface defined for the object, the message handling system of VML tries to delegate the message to another object by executing the method implementation given in a **NOMETHOD** clause. This permits for example to implement different inheritance strategies.

The class system in VML is organized in four levels: the individual *object level*, the *application class level*, the *metaclass level*, and the *root level*. At the metaclass level the system administrator may define new metaclasses and, thus, enhance the modeling capabilities of the predefined kernel model. Built-in classes at the root level provide for the basic and system inherent capabilities like object creation, object deletion, and object storage.

4.2 ISA in the VODAK Data Model

Based on the need to model real-world applications we discuss two examples of ISA specialization relationships which were implemented in VODAK using the metaclass system.

(a) *Role Specialization*, wherein real world objects may appear in different roles, e.g. a person may appear in the role of a student or an employee.

(b) *Category Specialization*, where provision is made for real-world objects being categorized into disjoint sets with respect to some specific aspect, e.g. parts may be categorized into simple parts and composite parts with respect to their complexity.

In the terminology of the previous sections, role specialization is based on extrinsic criteria and permits overlapping. When declaring it, one must specify whether or not the specialization should be restricted to a single general class. An example with multiple general classes is: from the more general classes Student and Employee, define the specialized class Employed Student. Category specialization requires disjointness but not covering. The membership criterion is solely based on intrinsic characteristics.

The strategy to implement these kinds of ISA specialization takes full advantage of the Metaclass concept. At the meta level there exist metaclasses Role-Specialization-Class, General-Category-Class, and Category-Specialization-Class. For role specialization, each general class is instance of the general VML Kernel-Application-Class metaclass. Each specialized class is connected, via *rolespec* relationship edges, to the general class and is instance of the Role-Specialization-Class metaclass. A number of public methods are passed by inheritance from these metaclasses to their instances, namely the role specialization classes, as well as to their metainstances, namely the instances of the role specialization classes. These methods allow, among other tasks, to create, delete and modify role specialization classes and their instances consistently and to interrogate the structure of classes and instances, finding for example, in the case of multiple specialization, which general classes a given class specializes and which general instances correspond to a given specialized instance. Furthermore a **NOMETHOD** clause allows to specify a particular inheritance behavior for instances of the role specialization class. The body of this clause is executed whenever no appropriate method is found for the receiver object of the method.

Example 4. More specifically the (interface) definition of the metaclass Role-Specialization-Class (RoleSpec, in abbreviated form) is as follows.

```
CLASS RoleSpec METACLASS Metaclass
  INSTTYPE OBJECTTYPE      // instance type of the specialized classes (inline definition)
    PROPERTIES   roleSpecofClass: OID;
    METHODS      roleSpecof(c: OID);        // sets roleSpecofClass
                 roleofClass(): OID;        // returns OID of general class
  INSTINSTYPE OBJECTTYPE // type of the instances of the specialized classes
    PROPERTIES   roleSpecofInstance: OID;
    METHODS      roleSpecof(i; OID);        // sets roleSpecofInstance
                 roleofInstance(): OID;     //returns OID of general instance
    NOMETHOD       // inherit current method call to the object roleSpecofInstance
END;
```

An application class, e.g. Employee being a role specialization of a class Person, is then defined as follows.

```
CLASS Employee METACLASS Metaclass
  INSTTYPE OBJECTTYPE Employee_type
                // application specific type of the instances of the application class
  INIT SELF→roleSpecof(Person)
END;            // initializes this class as specialization of the class Person ■
```

For category specialization, the general classes are instances of the General-Category-Class metaclass. Each specialized class is connected by a *catspec* relationship edge to the general class and is instance of the Category-Specialization-Class metaclass. The same powerful mechanism of inheritance of public methods is provided.

5 Design and Implementation of Advanced Querying Assistance

The VODAK DBMS provides a SQL-like query language named VQL [10]. For query processing SQL style queries are translated to an internal algebraic representation, to which rule-based transformations, e.g. for the sake of query optimization are applied [1,5]. For our purposes we will adopt this algebraic approach for user-oriented query modification. After a short review of algebraic query representations in object-oriented query algebras and the available rule specifications, we will give examples, how a non-conventional context is represented and exploited within this algebraic approach.

5.1 Query Algebras and Rule Systems

For the presentation in this paper we need to introduce only two basic query algebra operators of the object-oriented query algebra that is used for the internal representation of VQL-queries in VODAK. The algebraic operators are applied to complex values built up from tuple and set type constructors over atomic domains.

The first operator is needed to select from a set C of complex values, which is for example computed as the extension of a class, a subset according to a condition

SELECT(x: C, cond(x)) = { x ∈ C | cond(x) }

The second operator we consider is needed to apply a function iteratively to a set of complex values.

MAP(x: C, expr(x)) = { expr(x) | x ∈ C }

In order to use semantic knowledge of methods for query transformation we extend the data model language by a rule specification mechanism. Rules are then given in the interface of an object type by a clause of the form

RULES *name*
  V_1: dom_1; ... ; V_n: dom_n; {*cond*(V_1,...,V_n): $expr_1(V_1,...,V_n)$ <—> $expr_2(V_1,...,V_n)$};

where $V_1,...,V_n$ are the pattern variables, which stand for VML expressions of type dom_1, ... dom_n, $expr_1$ and $expr_2$ stand for (equivalent) VML expressions containing the pattern variables and *cond* stands for a Boolean VML expression containing the pattern variables.

Example 5. An example of such a rule in a person database is

RULES family
x : Person; {**true**: x→grandfather() <—> (x→father())→father()};

Such a rule can then be used to transform the algebraic representation of a query in the following way

MAP(x: Person, (x→father())→father()) ⇒ **MAP**(x: Person, x→grandfather()) ■

Including rules into object-type definitions in this way is a non-trivial enhancement of object-oriented data models towards the ADT (abstract data type) paradigm. It is not necessary that the rules provide a complete behavioral specification of the types of database objects. A fundamental point is that their declarative style makes them easier to understand and to serve for documentation purposes, for guiding the procedural development of the implemented code, and for testing its execution. The backbone for method execution is still their procedural specification. One difficult question that remains is how to maintain consistency between the declarative specification and the operational specification of a method or property. As a consequence of making rules part of the object type specification, rules can be inherited which is crucial for efficient design of rule systems.

The technique for transformation of the algebraic representation of a query by using rules specified in application schemas was implemented for VODAK in the context of query optimization. The main tool used was the Volcano optimizer generator [5]. We now show how this technique can be used for the purpose of user-assisting query modification.

5.2 Representation of Non-conventional Context in Database Schemas

The factual database and the schema (represented in the data dictionary) are already part of the context. As the assistance to the user must be tuned according to the user, characteristics of the user are represented in user models, which again take the form of classes.

Additional knowledge about the user may now be provided by specifying rules in the user model. The main idea for the implementation of query modification presented here is that the parameters given for the algebraic query operators are not taken for granted but need interpretation according to the context described. This interpretation is then provided by the rules of the user model.

Thus in a first step we produce a *context-sensitive algebraic representation* of the query. Assume the context is given by a particular user u and the query has the form

SELECT(x: C, cond(x))

Taking the parameters not for granted means, e.g., that we are not sure whether the user u really meant class C when he specified it. Thus we replace the plain class specification C by a user-sensitive method call

u→class_dom(C)

where the method class_domain represents the system model of how the user interprets a class specification C. The same is done for all other parameters appearing in the parameters of query operators, i.e. select and map operations are transformed to

```
SELECT(x: u→class_dom(C), u→bool_expr(cond(x))),
MAP(x: u→class_dom(C), u→int_expr(expr(x))),
```

provided that expr(x) evaluates to an integer value. The methods boolean_expr and integer_expr are used in the same way as class_domain to describe how the user interprets the particular expressions in the arguments.

Methods needed for transforming a query into the context-sensitive representation are given in the following object type declaration:

```
OBJECTTYPE QueryContext
  INTERFACE
  METHODS    class_dom(c: OID): OID;
             int_expr(o: OID, p: VML_PROP): INT;
             int_equal(i: INT, j: INT): BOOL; ...
  RULES  class_dom_rule c: OID {true: class_dom(c) <—> c};
         int_expr_rule o: OID, p: VML_PROP {true: int_expr(o,p) <—> o.p;}
         int_equal_rule i: INT, j: INT {true: int_equal(i,j) <—> i==j;} ...
  IMPLEMENTATION
  METHODS    class_dom(c: OID): OID {RETURN c };
             int_expr(o: OID, p: VML_PROP) {RETURN o.p};
             int_equal(i: INT, j: INT): BOOL { RETURN i==j }; ...
```

This object type is system-defined and is needed by a system module that will produce the context-sensitive algebraic representation of the query. Note that the implementation of the methods is simply identical to what would have been expected from the original expressions. On the basis of this type now a user class can be defined as follows:

```
CLASS User
  INSTTYPE OBJECTTYPE SUBTYPEOF QueryContext
  INTERFACE
  RULES    // additional rules on class_dom, int_expr, int_equal etc.
```

This class makes the methods inherited from the object type QueryContext nondeterministic, by allowing different interpretations of the same expression. Otherwise the rules do not differ at all from those used, e.g. in query optimization. Concrete rules will be given in the examples of the next section.

We remark that using the object type QueryContext as indicated above is a typical example for the application of the dual model approach of VML. It can be attached to other classes like those referring to characteristics of category specialization, or application classes, in case of application-dependent rules.

5.3 Application of Nonconventional Context for Query Modification

In this section we give three typical examples of how to use the rule mechanism for query transformation.

Example 6. Assume a user u of class User in the flight database of Section 2.3 issues the following query

```
SELECT f.number FROM f IN Flight WHERE f.price=1000
```

The corresponding context sensitive algebraic representation produced by the system according to the definitions given section 5.2 is as follows.

```
MAP(x: SELECT(x: u→class_dom(Flight), u→int_equal(x.price,1000)),
                                                u→int_expr(x.number))
```

Assume, that the following simple user model is given

```
CLASS User
   INSTTYPE OBJECTTYPE SUBTYPEOF QueryContext
   INTERFACE
   RULES adapt_price
            u: User; f: Flight; p: INT;
            {true: u→int_equal(f.price,1000) <—> f.price>0.9*p and f.price<1.1*p};
```

That means that for class_dom and int_expr no rules are applicable, except the identity rules class_dom_rule and int_expr_rule, which are inherited from the system defined object type QueryContext. Applying these rules gives

```
MAP( x: SELECT(x: Flight, u→int_equal(x.price,1000), x.number)
```

However, for int_equal we have an applicable rule adapt_price specified in the class User. Applying this rule results in the following modified query

```
MAP( x: SELECT(x: Flight, x.price>900 and x.price<1100, x.number)
```

which eventually will be evaluated. ■

After discussing the approach of rule application in principle we come to the central examples involving ISA hierarchies. The rules relevant for these will be introduced at the same level at which the semantic relationships themselves are introduced, namely at the metaclass level.

Example 7. In Example 3, query (1), of Section 3.3, an applicable rule should express, roughly speaking: if the query refers to a class that is a category specialization, find the corresponding general class and, through it, each of the other specialized classes to be tried in the new query. In the notation described, this means that the conditional part of the rule determines if the originally indicated class orig and some other class alt are instances of the metaclass Category-Specialization-Class (CatSpec), determined by applying the method catofClass[2]. If the condition holds, then alt will replace orig in the modified query; u refers to the current user.

```
RULES catgen
         u: User; orig: CatSpec, alt: CatSpec
         {orig <> alt and alt→catofClass() == orig→catofClass():
               u→class_dom(orig)<—>alt}
```

This is a rule that involves a complex condition. Hence we illustrate the whole process how the rule is applied. Let the user pose the following query:

```
SELECT e.address FROM e IN Enterprise WHERE e.name=="Alpha"
```

The context-sensitive representation is then given as

```
MAP( x: SELECT(x: u→class_dom(Enterprise), u→str_equal(x.name,"Alpha"),
                                                u→str_expr(x.address)
```

For the method class_domain now the rule catgen is applicable. Enterprise and Gov_agency are different category specializations of Institution. Therefore the query is transformed to

```
MAP(x: SELECT(x: Gov_agency, u→str_equal(x.name,"Alpha"),
                                                u→str_expr(x.address)
```

which now produces the intended answer. Other rules can still be applied now, e.g. for relaxing the string comparison condition or enhancing the output of the query. ■

Example 8. In Example 3, query (2), of Section 3.3, which is a case of role specialization, an applicable rule expresses the following: if a query refers to p as a property

[2]) The method catofClass is analogous to the method roleofClass given in Example 4 in Section 4.2.

(respectively method) of a class c_gen, whereas p is a property (respectively method) of class c_spec and c_spec ISA c_gen, then make the new query refer to p of c_spec.

The conditional part of the rule finds whether there is a class c_spec that is a role of class c_gen, and whether the desired property p belongs to c_spec (rather than to c_gen, as in the original query). If the condition holds, then the new query will look for p at the level of the specialized class; o is an object identifier for an instance of class c_gen, and, again, u refers to the current user;

```
RULES rolegen
    u: User; c_gen: KernelApplicationClass; c_spec: RoleSpecializationClass;
    o: OID; p: VML_PROP;
    {o IN c_gen→allInstances() and p IN c_spec→properties() and
      c_spec→roleofClass() == c_gen: u→int_expr(o,p) <—> (o→roleof()).p}
```

Note how in the condition part of the rule methods from the system level (allInstances), from the metaclass level (roleofClass) and from the data dicitionary level (properties) are used. Also observe that in the domain declarations the class to which o belongs is not yet determined, thus the generic type for object identifiers **OID** is used at that place. ■

5.4 Notes on the Implementation

The features to implement user-oriented querying assistance in VODAK are rule specification in the object interface and rule-application in query processing. In the preceding sections we have described the integration of a rule-specification mechanism within the VML data model. These rule specifications will be inserted by the VML compiler into the data dictionary. VODAK uses a universal interface to the message handler which allows both the access to the database and to the data dictionary by method execution. Thus a component for performing rule application is able to access this information through this universal interface. This approach was already taken, e.g., in the query optimizer, where the *Volcano* optimizer generator [5] is used for rule-based optimization. Using the same tool for query modification suggests an interesting approach: define a cost model that relates an actual query with the relevance to the users' information needs. Alternatively a Prolog component could by used, similarly as in the NICE project.

Different developments of the VODAK prototype had been undertaken independently, particularly in the context of query processing, which were found to facilitate the implementation of the techniques described in this paper. An internal algebraic representation of VML expressions has been modelled to support the representation and manipulation of query expressions. An interface for exchanging VML expressions and messages between VODAK and the *Volcano* optimizer generator was designed. The extension of the VML data model with the rule constructs introduced in this paper completes the required set of tools.

6 Concluding Remarks

The VODAK data model has a particularly appropriate structure to accomodate user-assisting features in a natural way. Its definition in terms of metaclasses and classes has led to a modularized organization of properties and methods. This very same discipline is adopted as rules are introduced, so that they are grouped according to their degree of generality, and are inherited down or filtered away in order to take into account the needs of individual users. The architecture of the implemented VODAK database system iself was conceived to favor extensibility. We further recall that the extensions

to provide (1) general-purpose rules and (2) the user-assisting algorithms, builds on our previous experience, respectively, with the rule-based tool to optimize VQL query-processing [1] and with the prototypes developed as part of project NICE [8].

Future research will be concentrated in refining the user-modelling strategies and in extending user assistance to the more advanced components of VODAK, such as: the *multimedia environment*, where template rules may be applied to guide the choice of the best medium to communicate an answer; *cooperative activities* involving several agents, where multiple user (i.e. agent) models can be simultaneously activated and used to achieve a better coordination of the users' processes; *heterogeneous database integration*, where the usual transparency paradigm for combining different schemas would be modified, by letting the original user of one of the schemas benefit from opportunities arising from richer features of the other schemas.

Acknowledgement. We would like to thank Klemens Böhm for carefully reading the paper and giving many valuable suggestions for improving the paper.

References

1. Aberer, K., Fischer, G.: Object-Oriented Query Processing: The Impact of Methods on Language, Architecture and Optimization. Technical report 763, GMD-IPSI, (1993).
2. Chu, W. W., Chen, Q.: Neighborhood and Associative Query Answering. Journal of Intelligent Information Systems, 1, 1992, 355–382 (1992).
3. Cuppens, F., Demolombe, R.: Cooperative Answering: A Methodology to Provide Intelligent Access to Databases. In: 2nd International Conference on Expert Database Systems, L. Kerschberg (ed.), Benjamin/Cummings, 621–643 (1989).
4. Fikes, R. E., Nilsson, N. J.: STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. Artificial Intelligence 2, 1971, 189–208 (1971).
5. G. Gräfe, W. J. McKenna: "The Volcano Optimizer Generator: Extensibility and Efficient Search", *Proceedings of the 9th IEEE International Conference on Data Engineering*, pp. 209–218, Vienna, Austria, April 19–23, 1993.
6. Hull, R., King, R.: Semantic Database Modeling: Survey, Applications and Research Issues. ACM Computing Surveys, 19, 3, 201–260 (1987).
7. Hammond, P., Sergot, M.: Augmented PROLOG for Expert Systems, Logic Based Systems Ltd. (1984).
8. Hemerly, A. S., Casanova, M. A., Furtado, A. L.: Cooperative behavior through Request Modification. In: Proc. 10th Conference on the Entity-Relationship Approach, 607–621 (1991).
9. Hemerly, A. S., Casanova, M. A., Furtado, A. L.: Towards Cooperativeness in Geographic Databases. In: Proc. DEXA (1993).
10. Klas, W. et al: VML – The VODAK Model Language Version 3.0. Specification Document, GMD-IPSI (1992).
11. Klas, W., Aberer, K., Neuhold, E.J.: Object-Oriented Modelling for Hypermedia Systems using the VODAK Modelling Language (VML). To appear in: A.Biliris, T.Oszu (Edt.): Object–Oriented Database Management Systems. NATO ASI Series, Springer Verlag Berlin Heidelberg, December 1993.
12. Kracker, M.: Unschafes assoziatives Begriffswissen zur Unterstützung der Formulierung von Datenbankanfragen. Dissertation, Technische Universität Wien, April 1991.
13. Neuhold, E. J., Schrefl, M.: Dynamic Derivation of Personalized Views. In: Proc. 14th VLDB Conference, 183–194 (1988).
14. Stein, A., Thiel, U.: A Conversational Model of Multimodal Interaction. Technical Report GMD-IPSI (1993).
15. Stonebraker, M. R.: Implementation of Integrity by Query Modification. In: Proc. ACM SIGMOD International Conference on Management of Data (1975).
16. Webber, B. L.: Questions, Answers and Responses: Interacting with Knowledge Base Systems. In: On Knowledge Base Management Systems, M. L. Brodie, J. Mylopoulos (eds.), Springer, (1986).