Query by Class, Rule and Concept*

Martin Staudt Hans W. Nissen
Manfred A. Jeusfeld

Informatik V, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Germany
{*staudt,nissen,jeusfeld*}*@informatik.rwth-aachen.de*

Abstract

The ideal query language for a knowledge base will probably never be found: easy formulation and easy evaluation of queries are two conflicting goals. Easy formulation asks for a flexible, expressive language near to human language or gestures. The second goal requires an effective mapping to machine code which computes the correct answer in a finite number of steps. This paper approaches the problem by a query language with three faces. The first one projects queries to concepts of the knowledge representation language KL-One for easy formulation and readability. The second face presents queries as rules of a deductive database with fixpoint semantics. Finally, queries appear as classes whose instances are the materialized answer (view) to the query. The methods for maintaining and updating the views are compiled from their deductive interpretation.

*This work was supported in part by the Commission of the European Community under ESPRIT Basic Research Action 6810 (Compulog 2) and by the Ministry of Science and Research of Nordrhein-Westfalen. A version of this paper appears in *Applied Intelligence*, Special Issue on Knowledge Base Management.

1 Introduction

General and most often conflicting goals for a query language are expressive power, tractable processing, ease-of-use, and application program support. While knowing that tractability is crucial for any system we have to compromise on expressiveness. Analogously, a query language tends to become simpler and easier to use if implementation details are left out. Based on a broad study of existing proposals, we expect from a query language that it offers:

- a powerful language for selection, at least equal to query languages for relational databases (maybe enhanced with transitive closure or more general forms of recursion),
- support for (materialized) views, their exploitation in query optimization, and their incremental maintenance due to base data changes
- (almost) equal handling of derived objects and stored objects, i.e. nesting and update of views,
- compact definition of query facilities, e.g., parameterized views as abstractions for sets of queries, and
- support for the effective optimization of all above operations (possibly including semantic optimizations based on the object model).

In this paper, we propose an amalgamation of query language paradigms from object-oriented databases (*query by class*), deductive databases (*query by rule*), and knowledge representation languages (*query by concept*) that satisfies the above criteria to a large degree. The next section reviews related work on each of the aspects. Section 3 contains the exposition of our query language and discusses the interaction between the paradigms. Section 4 reports on the implementation of the query language in the system ConceptBase with special attention to a simple architecture and support for view update and maintenance. Finally, section 5 discusses fields of applications where our approach has been successfully used.

2 Three Views of Queries

The basic understanding of what constitutes a query strongly influences the design of a query language and a query processing concept, and thus which of the desirable features it can offer. In this section, we review query models followed in the areas of object-oriented databases, deductive databases, and AI concept languages which understand queries, respectively, as

- classes of answer objects manipulated by methods,
- deduction rules with parameterized answer attributes, or
- concepts with automatic classification through subsumption relationships.

It turns out that each of these approaches basically follows the idea to make queries the same kind of thing it also handles otherwise. Each is particularly suitable for certain application areas of query handling. Moreover, there are specific reasons for this suitability which lie in the very nature of the approaches. In the last subsection, we draw together these observations to come up with a list of requirements our query model has to satisfy.

2.1 Query by Class: The Object-Oriented Approach

Object-oriented databases (OODB) combine the paradigms of object identity, class membership, inheritance, and methods from object-oriented programming languages like SmallTalk [22] with the database functionalities of persistence, concurrency, security, and declarative querying.

Since classes are the mechanism for managing sets of objects it is natural to use them also for describing query results, seen as sets of answer objects. In a way, the query language SQL is an early approach to queries as classes. SQL queries describe the structure of the answer similar to the interface of a class. The relation of the answer to the database (WHERE clause) describes how to compute the answer, i.e. the implementation of the interface. Moreover, the usage of relations in the FROM clause compares well to assigning a class as a subclass of another class. Of course, SQL does not support object identity, inheritance, and methods.

Query languages for object-oriented databases are mostly dialects of SQL. They vary widely in their position on object identity of answer objects. The language MQL [33], for example, is purely values-based, i.e. the answers are complex terms witout object identity. An object preserving approach is followed by the COCOON system [37]: answer objects inherit their object identity from the object base. One can think of the object identifier as an attribute of the objects which never may be projected out during query expression. Recently, query languages with system-generated object-identity have been proposed. In XSQL [29] the designer of a query may provide a function which computes the identifier of an answer object based on the condition part of the query. This feature makes sense for queries whose answer is stored for later manipulation (so-called views). The identifier serves as the address of an answer object.

A general advantage of queries as classes is that answers can be managed the same way as the objects in the OODB. Methods for processing answer objects can be attached to query classes. The answer objects can be organized in generalization/aggregation hierarchies and reuse methods of OODB classes. Moreover, the integration of methods in OODB suggests an implementation approach to query evaluation/view maintenance based on triggers attached to specific object/query class subtypes.

2.2 Query by Rule: The Deductive Approach

A deductive database is a simple kind of knowledge base. It consists of a finite set of facts (extensional database), e.g. $L(c_1, ..., c_k)$ is a k-ary fact with predicate symbol L, and a finite set of deductive rules of the form

$$\forall\ x_1, ..., x_n\ L_1 \wedge ... \wedge L_m \Rightarrow L_c$$

The literals $L_1, ..., L_m$ may be positive or negative. As in logic programming languages, a query to a deductive database is a clause

$$\exists\ y_1, ..., y_k\ \neg Q$$

All variables occuring in a deductive rule or query have to be quantified. Deductive database systems (e.g., Lola [19] and EKS [41]) differ in the treatment of terms as predicate arguments and the handling of negation in deductive rules. In this paper, we restrict ourselves to the common denominator: negation in the conclusion literals L_c is disallowed and negated literals in the rule body must obey the stratification condition. The latter restriction outcasts rules of the form $A \wedge \neg B \Rightarrow B$. Few deductive databases like LDL [34] allow complex terms as arguments. Here, we assume that all arguments are either quantified variables or constants. This is also known under the term Datalog$^\neg$ in the database community.

With these assumptions the preferred interpretation is a certain minimal Herbrand model, the so-called perfect model. It can be efficiently computed by a fixpoint operator [12]. Queries in deductive databases look rather simple compared to object-oriented databases but they have some advantages. The membership condition is defined by the rules which have a matching conclusion literal. Thus, it is very easy to formulate parameterized versions of a query just by replacing one or more variables by a constant.

Implementation techniques for query optimization have been thoroughly investigated yielding algorithms for recursion optimization (magic sets [8], query/subquery approach [42]), and deductive integrity enforcement (e.g. [3], [35]). View maintenance algorithms (storing the result of a query and keeping it up-to-date with the database) have been less studied but are rather simple generalizations of integrity enforcement. Some work has been done on updating views in deductive databases, esp. by abductive reasoning [18],[30]. The notion of views gives a flexible way for defining access paths with the access path as rule body and the extension of the rule head as the index. It has been demonstrated that deductive rules are suitable for establishing the access to external relational and heterogenous databases.

Like relational databases, deductive databases are limited by their poor type system and restricted power of operations. The interactions with the system are more or less only updates and queries to the database leading to logic-based evaluations. Integrity constraints are used passively for update control, hardly for (query) optimization due to inherent undecidability [11]. Especially, there is no guidance which constraints are useful and simple enough for optimization. However, results from deductive integrity checking have been adopted by the active database community for automatic generation of view maintenance triggers [14] and are being integrated in the forthcoming SQL-3 standard.

2.3 Query by Concept: A Type Inferencing Approach

Starting with KL-One [9], concept languages in artificial intelligence have pursued the idea to define knowledge bases as type lattices of so-called concepts. A concept is constituted by axioms of the form

$$C(x) \Leftarrow\ \varphi(x) \qquad \text{or} \qquad C(x) \Leftrightarrow\ \varphi(x)$$

where φ is constructed from unary predicates referring to other defined concepts and binary predicates expressing relationships between concepts. Only one free variable x is permitted inside φ. Because of the restricted usage of logical connectives a dedicated syntax for concept expressions has been developed: the axioms are written as $C \sqsubseteq E$ or $C = E$ where E is an abbreviated form of φ with implicit variables.

An implication type sentence has to be read as a necessary condition for any x with $C(x)$. An equivalence denotes a sufficient and necessary condition. Concept languages have been widely used in terminological applications. The main purpose is to relate concepts to each other, e.g. to derive that $C(x) \Rightarrow D(x)$ holds for any model of the defined logical theory, i.e. the *subsumption* between C and D. There is a similarity with types in object-oriented databases and integrity constraints in deductive databases. However one should note that concept languages make statements for all possible models of the theory opposed to databases languages which usually only consider the current database state as their model.

Systems such as CANDIDE [4] and CLASSIC [2] use concept descriptions as a flexible query language for databases. Queries are just considered as a new concept which is positioned in the concept lattice by subsumption algorithms. Answers to the query are all instances of all subconcepts and some instances of the direct super concepts (satisfying the additional constraints of the new concept).

The benefit of concept lattices is that the search space of a query (i.e., a new concept) is greatly reduced. The exact placement of a query into the lattice opens the ability for intensional query answering: instead of enumerating all instances of the query the system answers with the names of the subconcepts of the query (slightly more complicated for the contribution of the direct super concepts). On the other hand, if the extension of a super concept is materialized then a query subsumed by such a concept may reuse the extensions as a fine range.

Tractable subsumption algorithms exist only for a few rather simple concept languages [16] (an important example of a tractable formalism is the entity-relationship approach [10]). Compared to queries as rules no parameterization of concepts has been studied so far.

2.4 Summary of Requirements

Our short review has shown that each approach has advantages supporting certain (but not all) functionalities of a desired query system. The question addressed in our approach is how to integrate the three approaches. The following basic observations underly our solution, the query class:

- Queries as classes and queries as concepts offer the same object-oriented structure for questions and answers; therefore, this will be our approach.

- To integrate objects and rules, we need an object model with a logic-based semantics that allows a two-way transition between both. Our model of choice, Telos (see section 3), goes even further by offering an object model with a standard deductive database semantics, it has a precise translation into facts, rules, and integrity constraints of a deductive database.

- Concept language subsumption algorithms can be expected to work safely and effectively only for a sublanguage of the query language offered, e.g., by a full deductive database. A way has therefore to be found to offer both aspects separately. Complete subsumption is only done on the sublanguage (subsequently called type system). As in programming languages and in constraint logic programming, the communication between both formalisms is one-way, that is, the type system influences query processing but not vice versa.

The next section defines our query language on top of a deductive object base model and investigates it under the three perspectives as classes, rules, and concepts.

3 Objects and Queries in O-Telos

The O-Telos data model [25] is a variant of the knowledge representation language Telos [32] with the property that its semantics is that of a deductive relational database. In the following subsections we introduce the basic properties of the data model and show how queries in each of their roles fit into this framework.

The syntax of query classes is a class definition with superclasses, attributes, and a membership condition. The interpretation is based on a deductive rule containing predicates for each of the clauses in the class definition. Integration with concept languages is viewed as the extraction of the "type" of a query class, i.e., the portion of the query which is representable as a concept expression.

3.1 Representing Objects and Rules

Deductive object bases can be defined as deductive databases with builtin axioms formalizing the three object-oriented abstraction dimensions of classification, aggregation, and generalization. We define a deductive object base as a triple $DOB = (OB, R, IC)$ where OB is the extensional object base, R and IC contain deduction rules and integrity constraints. The axioms are represented as predefined formulas in R and IC. As usual we demand that (OB, R, IC) is consistent, i.e. $OB \cup R \models IC$.

Let ID and LAB be sets of identifiers, and labels resp. An **extensional O-Telos object base** is a finite subset

$$OB \subseteq \{P(o, x, l, y) \mid o, x, y \in ID, l \in LAB\}.$$

The elements of OB are called **objects** with identifier o, source and destination components x and y and name l. Alternativly an object base can be represented as a semantic net. Objects of the form $P(o, o, l, o)$ (called *individuals*) are represented as nodes with label l. Instantiations of the form $P(o, x, in, c)$ (= "x is in class c") and specializations of the form $P(o, c, isa, d)$ (= "c is subclass of d") are represented as dotted resp. shaded directed links. All other objects $P(o, x, l, y)$ are called attributes and are drawn as labeled links between x and y.

The running example in this paper is based on the following scenario from a medical database: *Patients are persons, suffer from diseases and take drugs which have an effect on diseases.*

P(#Person,#Person,Person,#Person)
P(#Patient,#Patient,Patient,#Patient)
P(#Disease,#Disease,Disease,#Disease)
P(#Drug,#Drug,Drug,#Drug)
P(#Jack,#Jack,Jack,#Jack)
P(#Headache,#Headache,Headache,#Headache)
P(#Aspirin,#Aspirin,Aspirin,#Aspirin)

P(#isa1,#Patient,isa,#Person)
P(#suff,#Patient,suffers,#Disease)
P(#tak,#Patient,takes,#Drug)
P(#ag,#Drug,against,#Disease)
P(#suff1,#Jack,suffers1,#Headache)
P(#tak1,#Jack,takes1,#Aspirin)
P(#ag1,#Aspirin,against1,#Headache)

P(#in1,#Jack,in,#Patient)
P(#in2,#tak1,in,#tak)
P(#in3,#suff1,in,#suff)
P(#in4,#ag1,in,#ag)
P(#in5,#Headache,in,#Disease)
P(#in6,#Aspirin,in,#Drug)

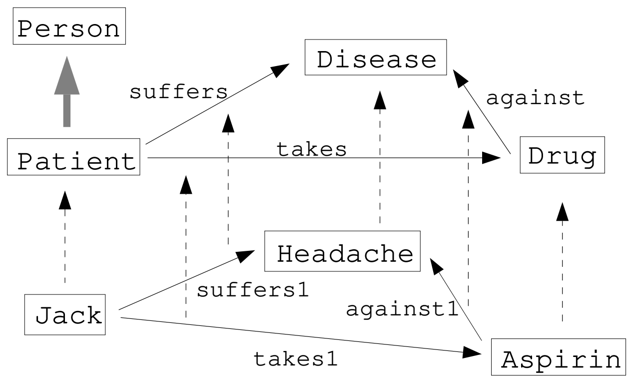

Figure 1: Tuple and semantic net view of deductive object bases

Fig. 1 shows the example graph derived from an object base OB representing this scenario. OB contains an object `Jack` which is instance of the class `Patient` and takes the drug `Aspirin` and suffers from `Headache`. By convention the object identifier of an individual with name `i` is written as `#i`.

A third view on deductive object bases results in a frame-based notation of objects which only relies on object labels. Around the name l of an object o we group the names of all other objects which have o as source component. The frame notation of the objects `Patient` and `Drug` is presented in Fig. 2.

O-Telos requires a set of 35 axioms and axiom schemata implemented as builtin deductive rules or integrity constraints of the deductive object base [25]. Here, we refer to a few important ones.

From the P-tuple representation three new predicates In, Isa and A can be derived.

$$\forall o, x, y \; P(o, x, in, y) \Rightarrow In(x, y)$$

$$\forall o, c, d \; P(o, c, isa, d) \Rightarrow Isa(c, d)$$

$$\forall o, x, l, y, p, c, m, d \; P(o, x, l, y) \wedge P(p, c, m, d) \wedge In(o, p) \Rightarrow A(x, m, y)$$

These literals are used when defining the complete axiomatization but also when specifying deductive rules and integrity constraints. Like most OODB and semantic data models, O-Telos requires that

- instances of an object are also instances of its superclasses, which is realized as the predefined deductive rule: $\forall x, c, d \; In(x, c) \wedge Isa(c, d) \Rightarrow In(x, d)$

In the example object base the instantiation link between the `takes1` attribute of `Jack` and the `takes` attribute[1] of `Patient` satisfies another property of the language, the attribute typing axiom:

[1]The `takes` attribute is also called an attribute class of `Jack`'s `takes1` attribute, and `takes` is referred to as its category.

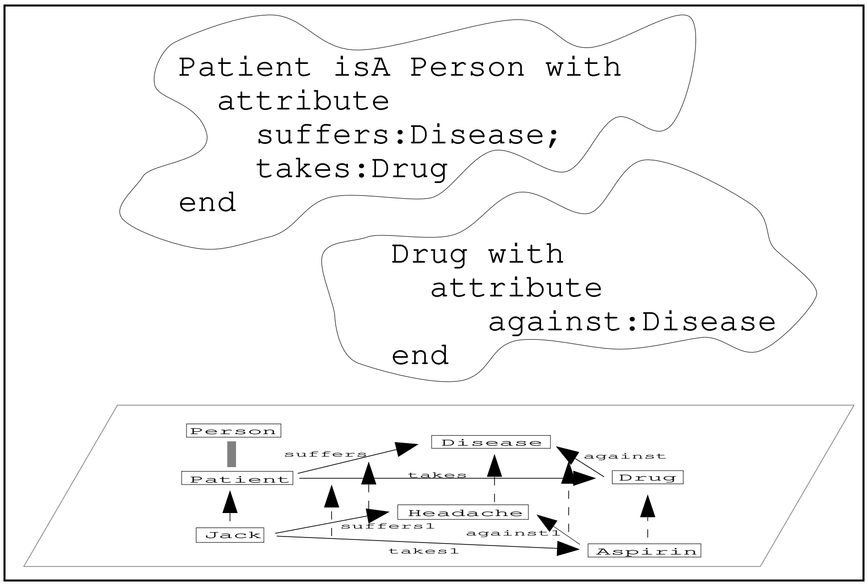

Figure 2: A frame view on deductive object bases

- instantiation relationships between objects imply instantiation relationships between their source respectively destination components.

A similar restriction holds for specialization links and allows a consistent way of refining attributes of classes by their subclasses. Thus, although attributes of an object are not explicitly inherited by subclasses they can nevertheless be instantiated by their instances.

Instantiation links between objects in OB constitute different layers within an object base. O-Telos does not restrict the number of layers; classes can be instances of other classes (metaclasses), these can be instances of metametaclasses, and so on. Closure is provided by reflexive objects `Object` and `Class` . The former contains all objects in OB as instances and the latter all objects which have instances, including itself.

In addition to the O-Telos axioms the sets R and IC contain application specific deduction rules and integrity constraints which are specified in a many-sorted first order language interpreted under closed-world assumption. Variable quantifications range over classes and are interpreted as instantiation relationships. In the frame syntax the literals are written as `(x m y)` where `x,y` are variables or constants (using object names instead of identifiers). If the second argument `m` is either `in` or `isa` then it is interpreted by the *In* or *Isa* literals, resp. Otherwise, it refers the A literal.

As an example a deduction rule can be defined that deduces for a given patient doctors who are suited to give him medical treatment. The constant `this` refers to the instances of the class which carries the rule.

```
Patient isA Person with
  attribute
    suited_doc:Doctor
  rule
    suited_rule:$ forall p/Doctor,d/Disease
                  (this suffers d) and (p specialist d)
                  ==> (this suited_doc p) $
end
```

Similarly one may impose an integrity constraint on `Patient` that all instances actually must have a filler for the `suffers` attribute.

```
Patient isA Person with
  constraint
    mustsuffer:$ exists d/Disease (this suffers d) $
end
```

The combined extensional object and rule/constraint base has the form of a deductive database with perfect model semantics. The variable ranges can be used to determine type errors inside the formula. This is a consequence of the attribute typing axiom.

3.2 Queries as Classes

The query language CBQL [38] represents queries as classes whose instances are the answer objects to the query. The predefined object `QueryClass` may have as instances so-called query classes which contain necessary and sufficient membership conditions for their instances (answers). These conditions can be used to check whether a given object is an instance of a query class or not. On the other hand, they can be used to compute the set of answer objects.

Query classes have superclasses to which they are connected by an isa-link. These superclasses restrict the set of possible answers to their common instances. Therefore, query class definitions are object-preserving. The set of answer instances of the query

```
QueryClass MaleOldPatient isA MalePatient,OldPatient end
```

consists of the intersection of all male and old patients existing in the OB.

Two different kinds of query class attributes can be distinguished. *Retrieved attributes* are already defined for one of the superclasses of the query class. An explicit specification of such an attribute in a query class description means that answer instances are given back with values for this attribute, similar to relational projection. In addition a necessary condition for the instantiation by an admissible value is included. The attributes of superclasses can also be refined, i.e. the target class is substituted by a subclass which results in an additional value restriction. If we assume a subclass `Antibiotics` of `Drug` the query class

```
QueryClass MaleOldAntibioticsPatient isA MalePatient,OldPatient with
  attribute
    takes:Antibiotics
end
```

has those male and old patients as instances who take antibiotics. In addition these drugs are included in the answer description.

Computed attributes have values derived in the query evaluation process. Neither the extensional object base contains this relationship between the answer instance and the attribute value, nor is it inferable by deduction rules. For prescribing how to deduce these new relationships by analogy to deduction rules and integrity constraints many-sorted first order logic expressions are admissible as building elements for query classes.

```
QueryClass WrongDrugPatient isA Patient with
  attribute
    wrong:Drug
  constraint
    wrongconstr:$ (this takes wrong) and
                  not exists d/Disease (this suffers d) and (wrong against d) $
end
```

As in section 3.1 `this` refers to the answer instances of `WrongDrugPatient`, namely all patients who take a drug which is against a disease they don't suffer from. The computed attribute `wrong` is identified with the variable of the same name within the formula. All deduced wrong drugs are part of the answer.

The logical expression in query classes descriptions can also contain arbitrary other constraints for the answer instances which would then work like the selection operation in relational databases.

In order to avoid the frequent reformulation of similar more specialized queries, attributes of query classes can be declared as instances of an attribute class *parameter*. Substitution of a concrete value for such an attribute or specialization of its target class by a subclass leads to a subclass of the original query which implies a subset relationship of the answer sets. If the `wrong` attribute were declared as parameter the expressions

```
WrongDrugPatient(Aspirin/wrong)
WrongDrugPatient(wrong:Antibiotics)
```

denote two derived query classes which restrict the answer instances of `WrongDrug-Patient` to those patients who take `Aspirin` or a drug of the class `Antibiotics` as wrong drug. For example, `WrongDrugPatient(Aspirin/wrong)` is a shorthand for

```
QueryClass WrongAspirinPatient isA Patient with
  constraint
    Aspiconstr:$ (this takes Aspirin) and
                 not exists d/Disease (this suffers d) and (Aspirin against d) $
end
```

Expressions denoting derived query classes are allowed to occur within the definitions of other query classes and objects and are managed as full-fledged objects, too.

3.3 Queries as Rules

In deductive databases queries are represented as intensional relations derived by a set of rules. We have shown above that O-Telos object bases are just special cases of deductive databases. Obviously it should be possible to extend this view to query classes.

The definition of a query class `Q` induces a so called *query literal* $Q(x, x_1, ..., x_n)$ whose arity depends on the number of attributes and parameters of `Q`. The first argument x of Q stands for the answer object identifiers. Query classes are transformed to rules concluding their corresponding query literals.

- For each superclass `C` of `Q` the body of this rule contains a literal $In(i, \#C)$ where $\#C$ is the object identifier of the class `C` and i is a new variable replacing `this` in the query class description.

- Attributes `a:C` of the first type (defined for a superclass or refined) result in a conjunction $In(v, \#C) \wedge A(i, a, v)$ where v is a new variable.
- Attributes `a:C` of the second type just require $In(v, \#C)$ where v is a new variable.
- The logical formula describing the derivation of attribute values and other additional restrictions is transformed straightforward by substituting the newly introduced variables v and i for their symbolic counterparts `a` and `this`, replacing all occuring object names by object identifiers and resolving typed quantifications.
- As a last step all generated expressions have to be linked by conjunctions. The new variables become arguments of the query literal.

Following these steps the example query class `WrongDrugPatient` is transformed to the following rule:

$$\begin{array}{ll} \forall i, v & In(i, \#Patient) \wedge In(v, \#Drug) \wedge \\ & \wedge A(i, takes, v) \wedge \neg \exists d\, In(d, \#Disease) \\ & \wedge A(i, suffers, d) \wedge A(v, against, d) \\ & \Rightarrow WrongDrugPatient(i, v) \end{array}$$

Each query class yields exactly one rule concluding its corresponding query literal. Hence with closed world assumption the rule body provides a necessary and sufficient characterization of class membership.

Since query classes may be superclasses or attribute value classes of other queries and their derived attributes may occur elsewhere, additional rules are needed based on the query literals.

$$\forall i, v\, Q(i, v) \Rightarrow In(i, \#Q)$$

$$\forall i, v\, Q(i, v) \Rightarrow A(i, wrong, v)$$

The deduction rule form of queries allows the application of the whole array of query optimization techniques known from the deductive database area [12].

3.4 Queries as Concepts

In programming languages objects (variables, functions, etc.) usually have types. Type constructors for tuples ([]), sets ({ }), lists etc. allow to handle objects of arbitrarily complex structure. In the same way most object-oriented databases distinguish between objects and values on the one hand and classes and types on the other hand. Objects have an unique identity and an assigned value of a certain type. Classes are object containers which collect objects of the same type (sometimes called member type).

O-Telos only provides the diction of classes where membership is constrained by the builtin axioms and by user-defined constraints. Nevertheless we can extract type information from classes. A class definition is divided into a 'clean' part, i.e. its type, and a 'dirty' part containing all aspects for which the type system is not powerful enough. The example class

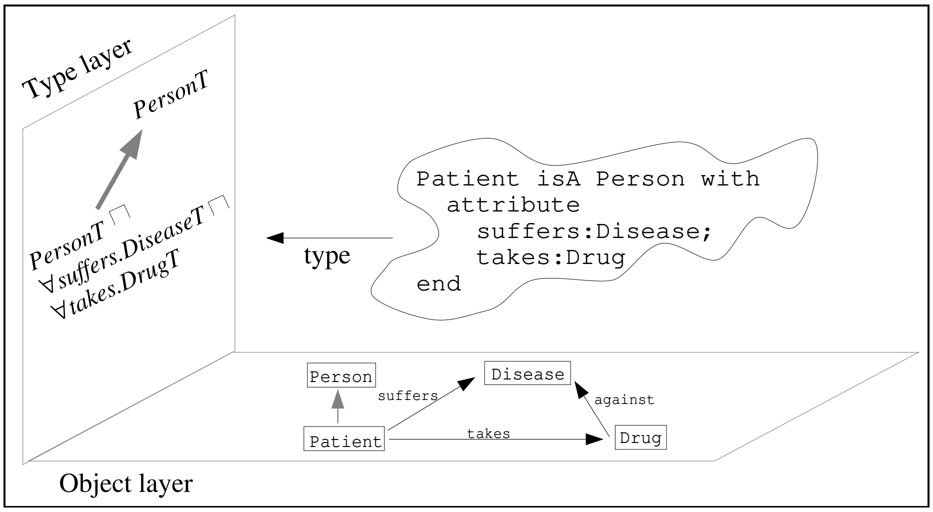

Figure 3: A type lattice for deductive object bases

```
Patient isA Person with
  attribute
    suffers:Disease;
    takes:Drug
end
```

carries the information that patients have attributes `takes` and `suffers`, each with certain value restrictions. In addition, patients have attributes (e.g. `name` and `address`) defined for the superclass `Person`. Under consideration that `takes` and `suffers` can have multiple fillers, a (member) type of `Patient` could be the following:

$$type(\texttt{Patient}) = [suffers : \{type(\texttt{Disease})\}, takes : \{type(\texttt{Drug})\},$$
$$name : string, address : type(\texttt{Address})]$$

Further readings on the differences between types in OODB's and concept languages are in [7]. It turns out that the latter allow more restrictions on objects than the type languages.

Following the syntax used in the concept language literature for our patient example, we get a member type $PatientT$ for `Patient`.

$$\forall suffers.DiseaseT \sqcap \forall takes.DrugT \sqcap PersonT$$

As a shorthand for $type(\texttt{A})$ we write AT for any class `A`. Fig. 3 presents the type information included in the frame-like O-Telos view as a layer orthogonal to the object base definition presented in 3.1. As in concept languages, this type layer is organized in a lattice structure. The 'type view' of deductive object bases introduced so far only used the $\forall$- and $\sqcap$- constructor for types and represented necessary conditions in the sense that e.g. every patient is also a person and that all attribute fillers for the `takes` attribute are drugs. Both conditions are guaranteed by O-Telos axioms. In a next step we allow other concept language constructors; for example, the entity relationship approach employs atmost ($\geq n\, r$) and atleast ($\leq n\, r$) cardinality restrictions for attributes r. Syntactically, this can be represented by introducing new built-in attribute categories `atleastn` and `atmostn`.

Type restrictions may be interpreted as sufficient membership conditions for their instances which in concept languages leads to the distinction between (defined) *concepts* and *primitive concepts* (the latter with only necessary conditions)[6]. The simple class definitions we had until now have primitive concepts as their types. Primitive concepts (as e.g. $PatientT$) are directly subordinated under their corresponding type expression in the lattice which exactly represents the interpretation as necessary conditions. In fact we have for `Patient`:

$$PatientT \sqsubseteq \forall suffers.DiseaseT \sqcap \forall takes.DrugT \sqcap PersonT$$

Defined concepts are related to their defining expression by equality. At the object layer we introduce a class `Concept` that includes all those classes that have a (defined) concept as their type.

The class `DrugPatient` contains all those patients that take atleast one drug. Since we want this to be interpreted as a sufficient condition for being instance of `DrugPatient` it is defined as instance of `Concept`.

```
Concept DrugPatient isA Patient with
  atleast1
    takes:Drug
end
```

At the type layer this corresponds to

$$DrugPatientT = PatientT \sqcap \geq 1\, takes$$

The definition is nearly[2] the same as defining `DrugPatient` as a query class:

```
QueryClass DrugPatient isA Patient with
  attribute
    takes:Drug
end
```

The concept class describes the same set of instances as the query class provided no constraints are used in the latter. In so far we can say that query classes as other classes can be separated into parts that can be covered by the underlying type language ('clean') and into 'dirty' parts that can't be. Instances of `Concept` can ve viewed as special query classes which just have a 'clean' part corresponding to a defined concept in the type language.

With the introduction of additional type constructs and their corresponding object counterparts we can apply the well understood inference algorithms for concept languages [23] at the type level of a deductive object base to infer new information at the object level. For example, if we define a class `AntibioticsPatient` just as a specialization of `Patient` with a value restriction for `takes` to drugs of the class `Antibiotics` with the additional requirement of at least one filler for `takes`, we can nevertheless place `AntibioticsPatient` automatically with respect to the isa-hierarchy below `DrugPatient` in the OB applying the rule

$$type(\texttt{ A }) \sqsubseteq type(\texttt{ B }) \Rightarrow Isa(A, B)$$

[2] apart from the projection on the `takes` attribute included in the query class description.

for each instance `B` of `Concept` and each object `A` where $\sqsubseteq$ denotes the subsumption relationship on the type lattice.

In the example a subclass relationship between `AntibioticsPatient` and `DrugPatient` can be concluded from

$$\begin{aligned} type(\texttt{AntibioticsPatient}) &\sqsubseteq PatientT \sqcap \forall takes.AntibioticsT \sqcap \geq 1\, takes \\ &\sqsubseteq PatientT \sqcap \geq 1\, takes \;=\; type(\texttt{DrugPatient}) \end{aligned}$$

provided that $AntibioticsT \sqsubseteq DrugT$. It follows from these considerations that the subclass hierarchy in the object base reflects the subsumption hierarchy at the type layer by deduced *Isa* relationships for concept classes. In the following we will concentrate on the relationship between concepts at the type layer and query classes.

The more complete the type language is, the more conditions of a query class `QT` can be expressed in a concept expression CT. With such clean parts one can perform a classification along subclass hierarchies. For the exact type QT of `Q` which cannot be described at whole within the type language in general (e.g. in the presence of constraints) we get the relationship $QT \sqsubseteq CT$. The classification process starts with all directly mentioned supertypes S_iT in CT and then tries to determine wether CT is subsumed by any defined concept DT below S_iT (i.e. $CT \sqsubseteq DT$). From this we can especially conclude that $QT \sqsubseteq DT$ holds at the type layer and subordinate `Q` below `D` in the isa-hierarchy at the object layer. This subsumption test is repeated for all subconcepts of DT etc. until the most specific subsumer(s) T_iT of CT and hence of QT are found. A subsequent deductive query evaluation process would then work on a restricted searchspace (i.e. consider the instances of T_i instead of S_i). Concepts that are subsumed by CT do not apply as sets of answer candidates since their specific further restrictions can contradict the 'dirty' parts of `Q` not covered by CT.

If `Q` however doesn't contain any 'dirty' parts ($QT = CT$) we can gain a significantly improved evaluation since then we can respect that QT itself eventually subsumes other defined or primitive (i.e. types of regular classes) concepts. If there is any defined concept DT below the S_iT and $DT \sqsubseteq QT$ then every instance of DT is definitely an instance of QT resp. `Q`. The same holds if there is a primitive concept subsumed by QT. In order to get an answer to `Q` one can just take the union of all instances for these subsumed concepts and only check all additional instances of the direct supertypes of QT by the deductive query evaluator for membership in `Q`.

3.5 Discussion

Query classes have been presented in the framework of the O-Telos data model. The main idea is to represent queries as classes, i.e. as special objects that can be placed in the network of objects constituting the object base. Instances of query classes can be understood as answers for the query. We showed that we can establish from this object-oriented view of queries a deductive view as rules on the one hand, and a concept view on the other hand. As a result we reach a high flexibility for expressing queries and inherit advantages of all three views.

4 Query and Update Processing in ConceptBase

From an implementation viewpoint, query classes can be understood as abstract data types on which the following kinds of operations must be supported:

- query evaluation,
- update processing,
- integrity checking,
- and view handling.

View handling includes two aspects namely maintenance of views (especially when they are materialized) and propagation of updates on views to stored base data. Integrity checking can be considered a special case of view maintenance.

In this section, we describe the implementation of query classes offered by the deductive object management system *ConceptBase* [24]. Subsection 4.1 shows the general system architecture and subsection 4.2 sketches the main aspects of query processing. Subsections 4.3 and 4.4 explain the view maintenance and view update mechanisms based on ECA rules.

Like in a couple of active or object-oriented databases, the basics of all these algorithms come from the deductive approach using the interpretation of query classes as rules. Due to the deductive database semantics of query classes this is more direct in our case than in most other OODB's. However, the other two aspects of query classes, namely their interpretation as classes and as concepts, allow further optimization and answer presentation techniques over and above those of deductive relational databases. These are discussed in subsection 4.5.

4.1 Implementation Architecture

Many existing systems employ completely separate subsystems and inference mechanisms for each of the operations above. An important goal of the implementation is to minimize the number of system components realizing these facilities in order to reach a modular architecture. Figure 4 shows the main system components involved in query and update processing. ConceptBase offers two main methods ASK and TELL, implemented by the system components `QueryProcessor` and `UpdateProcessor`. The controlling module `Q/A-Manager` of the `QueryProcessor` initiates the transformation (`Q/R-Compiler`) and evaluation (`QueryEvaluator`) of incoming queries and delivers answers back to the asking clients. In order to reuse stored answers (materialized views), the `Q/A-Manager` is supported by the controlling module `ViewManager` of the `UpdateProcessor` which collects all knowledge about materialized views or more general about views that are relevant (e.g. displayed on a screen) for some client.

The module `ViewManager` controls the processing of incoming update requests. We don't discuss in detail the TELL operations on extensional data, e.g. just storing a new instance of a class or asserting new attribute fillers, but concentrate on their effect on intensional data and on processing updates on views. The `ViewManager` employs the `ViewMonitor` in order to detect the change of a view as a consequence of the update. An update on a view is propagated to the module `ViewUpdate` to translate the view update to updates on base data. The `ViewManager` is also responsible for processing the consequences of an update request, i.e., rejection if integrity constraints are violated, materialization of view changes, and updating the object base. The following subsections describe the components of Fig. 4 in more detail.

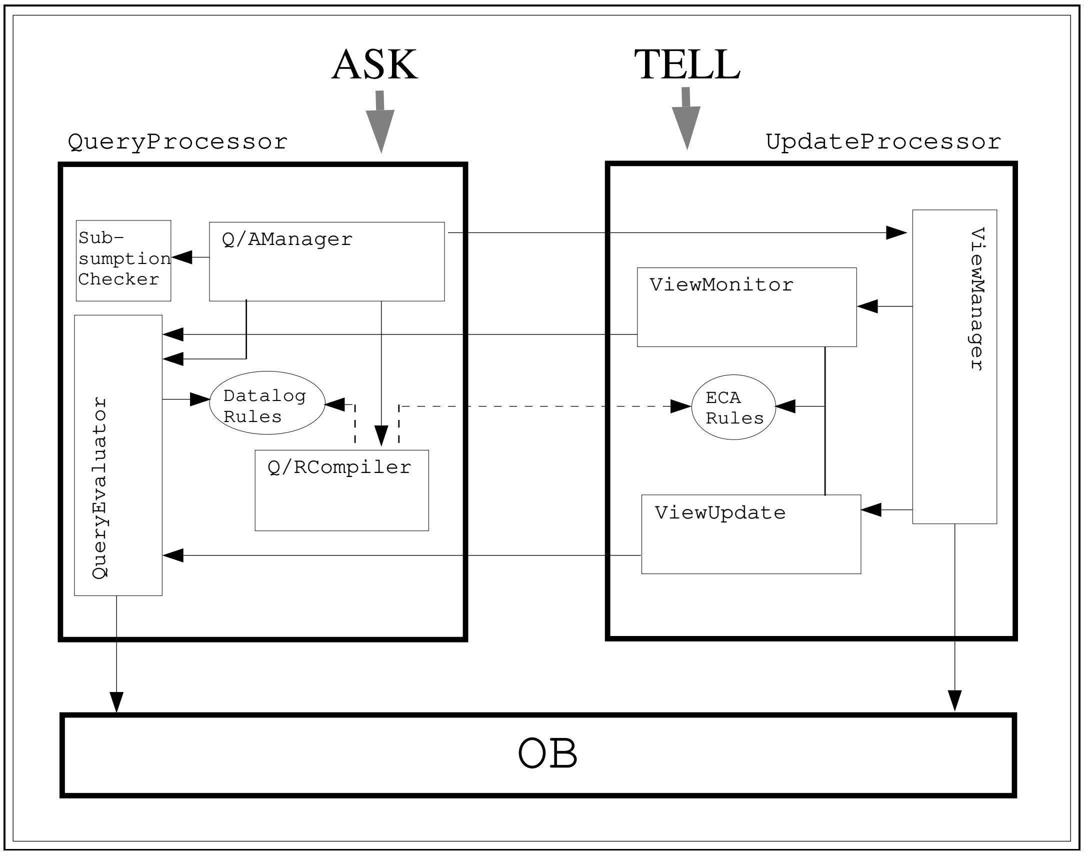

Figure 4: Query and update processing in ConceptBase

4.2 Query Processing

Query Compilation and Evaluation. The query processor provides subcomponents for the compilation and evaluation of queries. Strictly speaking query classes just serve for representing queries. On the other hand the rule perspective of query classes suggests to treat queries, integrity constraints and deduction rules completely in the same way. Thus, ConceptBase utilizes query classes to cover both integrity constraints and deduction rules. In subsection 4.3 we introduce special query classes for preserving the object base integrity.

Query classes that describe sets of objects important for administration purposes or that are frequently used by certain applications are called **views**. Sometimes these views are materialized, i.e., their answers are permanently stored.

The first task of a query compiler starting from a query description `Q` is to generate code for an efficient evaluation. We already presented the transformation of query class descriptions into rules. This intermediate representation can be compiled to sets of Datalog$^\neg$ rules utilizing the algorithm in [31] for translating general logic programs.

Our example query class `WrongDrugPatient` results in the rules

$$\begin{aligned} WrongDrugPatient(p,w) \; &:- \; In(p,\#Patient), In(w,\#Drug), \\ & \quad\;\; A(p,takes,w), \neg WDPH(p,w). \\ WDPH(p,w) \; &:- \; In(d,\#Disease), A(p,suffers,d), \\ & \quad\;\; A(w,against,d). \end{aligned}$$

The second Datalog$^\neg$ rule is an auxiliary rule concluding a new literal $WDPH(p, w)$ and results from resolving the negated existential quantification.

Performing an ASK operation means compiling one or more query classes into such a set of rules and then computing their answer instances in the current object base. There are different possibilities for evaluating these rules. A very obvious one is to simply apply a Prolog-like backward evaluation. Another one would be to use a bottom-up fixpoint procedure. On the other hand further compilation and optimization steps could follow. For efficient evaluation the Datalog$^\neg$ rules are translated into object algebra expressions [40] directly accessing the object store.

From Query Classes to ECA rules. The query evaluation component also supports the process of propagating explicit updates to changes of implicitly given knowledge defined by views. Since we are interested in *incremental* changes of derived views rather than a complete recomputation, the usage of the same general code compiled from query classes intended to be used for regular ASK-operations would lead to inefficiencies. Instead, the query compiler specializes for this purpose the original rules with respect to updates on literals occuring in the rule body. Each specialization yields a unique query literal with update parameters, supplementing the answer arguments similar to rules resulting from parameterized query classes. This specialization algorithm [26] follows the concept of relational simplification.

If the example rule for `WrongDrugPatient` is specialized with respect to a deletion of a disease a patient is suffering from, i.e. $Delete(A(p, suffers, d))$, the effect on the Datalog$^\neg$ rules is as follows:

$$\begin{aligned} WDP_1(p, w, d) \; :- \;\; & In(p, \#Patient), In(w, \#Drug), \\ & A(p, takes, w), A(w, against, d). \end{aligned}$$

The query $WDP_1(p, w, d)$ (always evaluated with instantiated parameters p and d) computes all drugs w which have to be inserted together with patient p into the view `WrongDrugPatient`, since p doesn't suffer from disease d any longer. Candidates for w are only those drugs that are against d.

A specialization concerning the insertion $Insert(A(p, takes, w))$ of a new drug a patient takes leads to

$$\begin{aligned} WDP_2(p, w) \; & :- \;\; In(p, \#Patient), In(w, \#Drug), \neg WDPH_1(p, w). \\ WDPH_1(p, w) \; & :- \;\; In(d, \#Disease), A(p, suffers, d), A(w, against, d). \end{aligned}$$

In this case we get an insertion of p and w into the view `WrongDrugPatient` if there is no disease of p for that w is against.

To directly support view maintenance and view update mechanisms the `Q/R-Compiler` generates event-condition-action rules (ECA rules) [15] of the form

```
ON  <event>
ASK <query>
DO  <action>
```

An event is a direct or induced update on one of the base literals A, In, IsA or on a query literal. The condition of an ECA rule consists of a query which has to be evaluated under the respective parameter settings caused by the update. The action (a new induced update) is executed with the solutions of this query.

ECA rules for the updates $Delete(A(p, suffers, d))$ and $Insert(A(p, takes, w))$ effecting `WrongDrugPatient` are for instance

```
ON  Delete(A(p,suffers,d))           ON  Insert(A(p,takes,w))
ASK WDP1(p,w,d)                      ASK WDP2(p,w)
DO  Insert(WrongDrugPatient(p,w))    DO  Insert(WrongDrugPatient(p,w))
```

4.3 Incremental View Maintenance

TELL operations on an object base results are realized by insertions or deletions of objects. The resulting changes often affect intensional knowledge defined by views and deduction rules. This holds especially for the case the derived knowledge is at least in part materialized. Thus, view maintenance is the process of efficiently propagating updates of base data to views in order to guarantee the correctness and completeness of potentially stored answers. Since integrity constraints can be interpreted as views containing the erroneous objects as answer instances, integrity checking is just a special case of view maintenance. Every insertion into such an integrity view signalizes the violation of the corresponding constraint.

The example constraint in subsection 3.1 can be written as

```
QueryClass suffersIC isA Patient with
  constraint
    ic:$ not exists d/Disease (this suffers d) $
end
```

This constraint corresponds to the query rule

$$\forall i, y \; In(i, \#Patient) \wedge In(y, \#Disease) \wedge A(i, suffers, y) \;\Rightarrow\; suffersIC(i)$$

Of course, the extension of *suffersIC* must be empty for any consistent object base.

As indicated in section 4.1 the `ViewManager` gets an update request from an external client and, in the case of an update of base data, decomposes it into updates on base literals and propagates them to the `ViewMonitor`. The task of the `ViewMonitor` is to detect all changes of views as consequences of the update request. In this section we describe the way the `ViewMonitor` detects the change of views using ECA rules. This method computes for the original update request all consequent updates in an incremental way. Each of them possibly causes the change of a view that can be efficiently checked using the simplified forms generated by the `Q/R-Compiler`. To reduce the management overhead specialized rules are only computed for relevant views, i.e. views that are e.g. assigned to be "firstclass" by the user explicitly. We distinguish two different kinds of actions: actions representing an update concerning a base or query literal (`Insert/Delete(_literal)`) and actions describing the violation of an integrity constraint (`Reject(_u,_reason)`). Figure 5 presents in a Prolog-style language the algorithm for an incremental processing of updates. ECA rules are stored as `eca(on(_event),ask(_condition),do(_action))`. We use the symbol '&' to indicate the concatenation of lists and three dots (...) to denote lists of variable length.

Input: update request of the form `Insert(_literal)` or `Delete(_literal)`, where `_literal` is one of the base literals `In, Isa,` and `A`.
Output: set of actions of the form `Insert(_queryliteral)` or `Delete(_queryliteral)` or `Reject(_update, _reason)`.

```
ProcessUpdate ([], []).
ProcessUpdate ([_update|_restUpdates], _updViews & _restUpdViews) :-
     FindECA (_update, _ecaList),
     ProcessECA (_ecaList, _newUpdates, _updViews),
     ProcessUpdate (_newUpdates & _restUpdates, _restUpdViews).
ProcessUpdate ([_update|_restUpdates], _updViews) :-
      ProcessUpdate (_restUpdates, _updViews).

ProcessECA ([], [], []).
ProcessECA ([eca(on(_event), ask(_query), do(_action)) | _restECA],
            _updates & _restUpdates, _updViews & _restUpdViews) :-
    Ask (_query, [_answer_1, ... , _answer_n]),
    InstantiateAction (_answer_1, _action_1),
    . . .
    InstantiateAction (_answer_n, _action_n),
    SortActions ([_action_1, ... , _action_n], _updates, _updViews),
    ProcessECA (_restECA, _restUpdates, _restUpdViews).
ProcessECA ([_firstECA| _restECA], _updates,  _updViews) :-
    ProcessECA (_restECA, _updates, _updViews).
```

Figure 5: View maintenance algorithm

For an update request `Upd` the algorithm starts with the call `ProcessUpdate([Upd], _updViews)` where `_updViews` is a free variable. It returns in `_updViews` the actions that have to be performed on views. If this list contains an action `Reject(_v,_reason)`, the update request `Upd` or an update deduced from it violates an integrity constraint. We only give a brief description of the algorithm and illustrate it using examples.

The procedure uses three lists: a list of remaining updates to be processed, a list of computed updates of views and a list of matching ECA rules for the actual update.

The algorithm collects in the first step all ECA rules whose event parts match the first update (denoted by U) in the update list. If no ECA rules exist for U, this update forces no further changes and will be deleted from the update list. If ECA rules exist, the `QueryEvaluator` checks the condition of the first rule on the object base $U(OB)$ (where $U(OB)$ denotes the result from performing U on the actual object base OB). Every solution of this query is used to instantiate the free variables of the corresponding action. In the case of constraint violation the query returns the reasons for the violation. In the case of a changed view the solutions describe the information to be inserted resp. deleted from the view, i.e. the incremental change to the view.

The procedure `SortActions` distributes the actions among the list of remaining updates and the list of updates of views. Actions like `Insert/Delete(_literal)`, where `_literal` is a view literal, for example, must be treated in a twofold way. First they represent an update of a view and will be added to the list of computed view changes. But they also represent an update that could force the change of other views, so they will also be added to the update list. For every `Delete(_literal)` this procedure also tests if `_literal` is still deducible in $U(OB)$, because the conditions of ECA rules are already satisfied if an update eliminates only one inference path to deduce `_literal`.

If this is done for all ECA rules matching the update U, the algorithm starts again with the remaining updates. The algorithm stops if the update list is empty, i.e. all consequences of the update request `Upd` are computed.

```
QueryClass SuitedDrug isA Patient with
  attribute
    suited_drug: Drug
  constraint
    suitedconstr: $ exists d/Disease (this suffers d) and
                  (suited_drug against d) $
end
```

To demonstrate the algorithm we assume the existence of two views: the query class `WrongDrugPatient` already defined in section 3.2 and the query class `SuitedDrug`.

This view computes for all patients the possible drugs against the diseases they suffer from. The `Q/R-Compiler` generates for each view a set of ECA rules where the condition consists of a query representing a specialized rule of the corresponding view. The rules generated to handle an update of the literal `A(x,suffers,y)` are as follows:

```
(1): ON  Insert(A(x,suffers,y))
     ASK WDP3(x,a1,y)
     DO  Delete(WrongDrugPatient(x,a1))

(2): ON  Delete(A(x,suffers,y))
     ASK WDP1(x,a1,y)
     DO  Insert(WrongDrugPatient(x,a1))

(3): ON  Insert(A(x,suffers,y))
     ASK SD1(x,y,a)
     DO  Insert(SuitedDrug(x,a))

(4): ON  Delete(A(x,suffers,y))
     ASK SD2(x,y,a)
     DO  Delete(SuitedDrug(x,a))
```

The object base should contain only the following instances of `Patient`. In this example we assume a class `Patient` without any integrity constraint.

```
Bill in Patient with
  takes
    bill_takes : Aspirin
  suffers
    bill_suff1 : Headache;
    bill_suff2 : Appendicitis
end

John in Patient with
  takes
    john_takes : Aspirin
  suffers
    john_suffers : Appendicitis
end

Fenta in Drug with
  against
    ag : Appendicitis
end

Appendicitis in Disease end
```

We describe the extensions of the two views by listing all solutions of the query literals:

```
WrongDrugPatient(#John,#Aspirin)

SuitedDrug(#Bill,#Aspirin)
SuitedDrug(#Bill,#Fenta)
SuitedDrug(#John,#Fenta)
```

For the update request `Delete(A(#Bill,suffers,#Headache))` the algorithm is called by `ProcessUpdate([Delete(A(#Bill,suffers,#Headache))],_updViews)` and works for our example object base as follows:

The first step collects the two ECA rules (2) and (4), which match the update. After that the `QueryEvaluator` checks the condition of rule (2) and returns the answer `WDP1(#Bill,#Aspirin,#Headache)`. This results in the following update of the view `WrongDrugPatient Insert(WrongDrugPatient(#Bill,#Aspirin))` which will be inserted into the list of remaining updates and into the output list. The condition of ECA rule (4) returns the answer `SD2(#Bill,#Headache,#Aspirin)` which leads to the action `Delete(SuitedDrug (#Bill,#Aspirin))`. This is also added to the update and the view list. Because there exist no matching ECA rules for both remaining updates, the algorithm terminates and finally returns `[Insert(WrongDrugPatient(#Bill,#Aspirin)), Delete(SuitedDrug(#Bill,#Aspirin))]` as the sulution. This list is passed to the `ViewManager` which controls the conclusive change of the materialization of the views.

4.4 View Update

Besides update operations on base data a second special case are TELL operations on views. One fundamental goal of views in databases is transparency: A user shouldn't have to be aware of wether the classes constituting his database subschema are virtually defined (views) or have a one-to-one correspondence to explicitly stored classes. This section deals with the problem of appropriately translating a request to update a view into updates of the underlying base data, i.e. data not defined by a rule. The main problem is to determine the 'best' realization of the update if a set of plausible explanations arise. Another important problem connected with a view update is the handling of hypotheses within an object base.

The view update problem becomes simple if we allow updates only on views where the propagation to base updates results in an unambiguous explanation. For an insert operation on a view defined using only the algebra operations intersection, selection, and projection (i.e. a view containing only retrieved attributes) we can always find an unique translation to a set of base updates because the corresponding query rule contains only a conjunction of literals. The insertion of exactly all these literals makes the query literal deducible. This leads to an unique realization of the view update where no hypotheses have to be introduced into the object base. Disjunction and free variables in the rule body lead to different possible explanations of an update.

To propagate view updates to base updates we do not use the (internal) rules generated to compute the extension of the view (see section 4.2) but a special kind of ECA rules. They can be generated at compile time and each of them describes a single abductive step. The action part of these ECA rules consists of a term describing the conjunction and/or disjunction of insertion and/or deletion updates in order to realize the update described in the event part. Because there is no condition necessary to describe the consistent application of the action part, we use the condition that will always be satisfied (`True`).

The restriction to handle only unique view updates is not very satisfactory since the majority of views can not be defined in a way where the update always results in an unambiguous explanation. Consider the example view `WrongDrugPatient` and the generated ECA rules for the insert operation:

```
ON  Insert(WrongDrugPatient(a,b))
ASK True
DO  AND [ Insert(In(a,#Patient)), Insert(In(b,#Drug)),
          Insert(A(a,takes,b)), Delete(WDPH(a,b)) ]

ON  Delete(WDPH(c,d))
ASK True
DO  OR [ Delete(In(s,#Disease)), Delete(A(c,suffers,s)),
         Delete(A(d,against,s)) ]
```

Because the computation rule in Datalog$^\neg$ format for this view uses an additional predicate (`WDPH`) in order to realize the negation (see section 4.2), a hierarchy of ECA rules is generated. It is easy to see that an update on the `WrongDrugPatient` view would not result in an unique explanation. In order to handle updates on such views, we propose a three step algorithm:

Generation: In the first step the algorithm generates a term consisting of conjunction and/or disjunction of updates of the form `Insert/Delete(literal)`. The generation procedure starts with the original update request and evaluates all matching ECA rules which lead to a disjunction of their action parts in the case of an insertion and a conjunction in the case of a deletion. The original update is replaced by this term and the procedure is recursively called for all updates appearing in this term. The procedure stops if there is no ECA rule matching any of the updates in the term. The output term represents possible (but not necessary consistent) ways to realize the original update.

Optimization: Since the generated term contains updates on literals occuring in integrity constraints this step uses these constraints on the one hand to enforce the consistency of the term (e.g. by introducing necessary additional updates) and on the other hand to simplify the term (e.g. by instantiation or restriction of variables or elimination of superfluous or inconsistent updates). A further optimization bases on the application of axioms of the O-Telos data model. If the generation of a consistent term is not possible, the procedure rejects the view update.

Reduction: In the conclusive step of the algorithm the term is reduced such that the resulting term consists of exactly one conjunction of updates and no disjunctions. The aim of this step is to eliminate disjunctions by selecting one of the disjunctive updates. This elimination procedure implements the strategy of choosing the most likely realization of an update request.

The reduction procedure uses a (system defined) priority order (like in [20, 21]) upon the literals as a first attempt for the elimination of disjunctions, e.g. we would prefer the deletion of an object's attribute to the deletion of the instantiation relationship of an object. Since further reduction requires information about the domain of interest, the application of user defined priority orders is possible. If this doesn't work, the user is asked to discriminate between different possibilities. In the case of deletions the range of variables is always finite, because the extensional object base is finite. To handle variables in insertions, we assume with every domain a set of an arbitrary number of distinct (Skolem) constants. We then select for every variable a new constant. We can ensure minimality if we impose a total order on every set of constants.

Input: update request of the form `Insert(_queryliteral)` or `Delete(_queryliteral)`
Output: term to describe the resulting updates on base data or the string `inconsistent view update`

```
ProcessViewUpdate(_update,_reducedTerm) :-
    BuildTerm(_update,_term),
    Optimize(_term,_optTerm),
    Reduce(_optTerm,_reducedTerm).

BuildTerm(_singleUpdate,_update) :-
    abducible(_update).
BuildTerm(_singleUpdate,OR/AND([t_1,...t_n])):-
    GetECA(_sigleUpdate,OR/AND([do_1,...do_n])),
    BuildTerm(do_1,t_1),
    ...
    BuildTerm(do_n,t_n).
BuildTerm(OR/AND([u_1,...,u_m]),OR/AND([t_1,...,t_n])):-
    BuildTerm(u_1,t_1),
    ...
    BuildTerm(u_n,t_n).
BuildTerm(_singleUpdate,'inconsistent view update').
```

Figure 6: Algorithm for view update

Figure 6 shows the implementation of this algorithm in a Prolog-style language. We only sketch the main procedure and the procedure realizing the first step. Since it is not clear how to translate recursive views we restrict ourselves to non-recursive rules, however allow hierarchical definitions of views. The procedure for step 1, realized by predicate `BuildTerm`, uses the predicate `abducible(_update)` to indicate that there exist no more matching ECA rules. The expression `OR/AND` indicate that the functor can either be an `OR` or an `AND`.

For an object base consisting of the view `WrongDrugPatient` and the example objects of section 4.3 the algorithm would produce the terms shown in figure 7 for the update request `Insert(WrongDrugPatient(#Jack,#Aspirin))`. The generation procedure first replaces the view update literal by the action part of the corresponding ECA rule. The result is term (1). The only term matching with an ECA rule is `Delete(WDPH(#Jack,#Aspirin))`. The replacement by the action part results in term (2). This term doesn't represents an unique update on base data, because it contains a free variable (`s`) and a disjunction.

Since the object base already entails the literal `In(#Aspirin,#Drug)` the corresponding insertion can be deleted. For every instantiation of variable `s` the object base doesn't entail the literals `A(#Jack,suffers,s)` and `A(#Aspirin,against,s)`, thus the deletions can also be eliminated. This leads to term (3).

Since the term (3) consists of exactly one conjunction, it represents the result of the algorithm. The conclusive updates to be performed on the object base are the insertion of the literals `In(#Jack,#Patient)` and `A(#Jack,takes,#Aspirin)`. It is easy to see that the resulting object base entails the literal `WrongDrugPatient(#Jack,#Aspirin)`.

To illustrate the idea behind the reduction procedure, consider the following term consisting of one disjunction:

```
                    Insert(WrongDrugPatient(#Jack,#Aspirin))
                                      |
                                      v
(1)  AND([ Insert(In(#Jack,#Patient)),Insert(In(#Aspirin,#Drug)),
           Insert(A(#Jack,takes,#Aspirin)),Delete(WDPH(#Jack,#Aspirin)) ])
                                      |
                                      v
         AND([ Insert(In(#Jack,#Patient)),Insert(In(#Aspirin,#Drug)),
(2)            Insert(A(#Jack,takes,#Aspirin)),
              OR([ Delete(A(#Jack,suffers,s)),
                   Delete(A(#Aspirin,against,s)) ]) ])
                                      |
                                      v
(3)                AND([ Insert(In(#Jack,#Patient)),
                         Insert(A(#Jack,takes,#Aspirin)) ])
```

Figure 7: Output for `Insert(WrongDrugPatient(#Jack,#Aspirin))`

```
OR([ Delete(A(#Bill,suffers,#Headache)),Delete(A(#Aspirin,against,#Headache))]).
```

In this situation we want the reduction procedure to eliminate the update `Delete( A(#Aspirin,against,#Headache))` because it is much more plausible to assume that `Bill` no longer suffers from `Headache` than to assume that `Aspirin` no longer has an effect on `Headache`. But to realize this preference order in a term processing procedure is very difficult and requires additional information about the domain of interest.

4.5 Query Optimization

The threefold view of queries as classes, rules, and concept opens various opportunities of optimizing the query evaluation process.

The axioms of the O-Telos data model can be used to get a significant reduction of the number of literals to be checked for queries and as a consequence also of the number of specialized forms and triggers (ECA-rules) for relevant views [25, 26]. Applying e.g. the axiom of section 3.1 which states that instantiation relationships between objects imply instantiation relationships between their source respectively destination components, all In-literals in the rule generated from the query class `WrongDrugPatient` can be dropped.

$$\begin{aligned} \forall i,v \quad & A(i,takes,v) \wedge \neg\exists d\, A(i,suffers,d) \wedge A(v,against,d) \\ & \Rightarrow WrongDrugPatient(i,v) \end{aligned}$$

The axioms guarantee that all variables are bound to the correct classes (as originally ensured by In) since they occur as source/destination components of attributes (resp. A-literals). In-literals of course can't be eliminated in general because they may express additional restrictions on variables. As a consequence triggers (ECA-rules) have to be generated only for insert and delete operations on the three occurences of A. The Datalog$^\neg$ representation reduces to

$$\begin{aligned} WrongDrugPatient(p,w) \quad &:- \quad A.\#tak(p,w), \neg WDPH(p,w). \\ WDPH(p,w) \quad &:- \quad A.\#suff(p,d), A.\#ag(w,d). \end{aligned}$$

where the new specialized forms $A.\#tak$, $A.\#suff$, and A.#ag of the literal A allow a more efficient access directly to the instances of the concerned attribute classes. This technique can be extended to views containing complex objects as well [27].

The rule view of queries allows the application of the standard deductive optimization techniques. One example is the application of the magic-set rewriting method [8] to the Datalog$^{\neg}$ rules generated from query classes. A bottom-up fixpoint procedure using the rewritten rules only computes information that is relevant to a given query.

From the concept view of queries the search space restriction for query evaluation by placing query classes in the object base using knowledge about subsumption relationships of their assigned type to the types of other classes can be extended to the reuse of precomputed resp. stored answers for other queries also called *materialized views*. This classification is realized by the system component `SubsumptionChecker` in figure 4. Especially the determination of such relationships between queries and views that are materialized promises increasing efficiency.

In this case we classify the type of a query against the types of these materialized queries. It should be obvious that we can only detect subsumption relationships between two queries where at least one has a completely defined concept as type. We can distinguish the same possible situations as in section 3.4 above. The most common situation is the following: A materialized query has a (defined) concept type that subsumes the type of the query to be evaluated. The set of objects relevant for the answer can then be restricted to the answers of the materialized queries and only some (hopefully inexpensive) selections will be necessary to get the final answer set. An efficient calculus for deciding subsumption between queries and views is presented in [5] using a query language very similar to O-Telos. The method is based on two specific concept languages with different expressiveness and respects semantic knowledge contained in class definitions.

4.6 Discussion

Following the comprehension of query classes as abstract data types we presented implementations for the following kinds of operations: query evaluation, integrity checking, update processing, and consistent view handling.

The uniform framework for these operations enables the reuse of the same components for different tasks and therefore leads to a clear architecture. We propose update-driven inference based on ECA-rules for both incremental view maintenance (including the special case of integrity views) and view update propagation.

Simplified forms of rules defining views lead to low implementation effort for view processing mechanisms. The threefold view of queries as classes, rules and concepts supports the application of well-known optimization techniques of each of these scopes to query evaluation.

5 Applications

The implementation of query classes in ConceptBase has turned out to be a useful basis for several application areas that can exploit the different facets of queries as classes, as rules, and as concepts. While some of these applications used the query language concept as is, others base extensions such as imprecise querying or access to external data stores on it. We briefly report on these experiences and point to more detailed publications by the corresponding ConceptBase users.

Software databases. For a long time *software reuse* has been regarded as a crucial goal in order to reduce both programming effort and program errors. The problem is how to organize a database of software modules which allows retrieval of reusable components. The combination of deductive and concept languages in query classes has been used to support similarity-based retrieval in algebraic specification databases [13]. Argument types and arity of operations are abstractly represented as query classes while more detailed semantic analysis is performed on retrieved answers with a specialized theorem prover. A second extension is that query relaxation techniques are incorporated to rank answers by similarity to the query rather than demanding precise instantiation.

Another aspect of software databases, *dynamic clustering*, has been investigated in [36]. A dynamic cluster is a collection of objects where the membership frequently changes. For example, a program object may belong for a certain short time to the cluster `Uncompiled`. Instead of procedurally assigning objects to such classes, a membership condition expressed as a query class automatically classifies objects into the correct clusters. Due to the deductive nature of query classes, the dynamic aspect of re-assigning objects is completely covered by view maintenance.

Security management. Security is becoming an increasing concern in many databases. The Group Security Model GSM [39] uses generalization and aggregation to define task-based access to parts of the object base (identified by classes). The question what information a certain task may access is encoded as a query class. A simulation toolkit based on the GSM studies security aspects of proposed information systems designs. Thus, the deductive view is applied for analysis while the object-oriented view and the availability of parameterized query classes are exploited for the precise and concise characterization of access right patterns.

Integration of heterogeneous databases. In many applications, the information to be processed is distributed in more or less autonomous information bases. Data may be replicated or even contradictory, the languages may be heterogeneous. Keeping the part databases at least partially consistent with each other is a combined view update and maintenance problem. Updates have to be propagated between the parts. The methodology proposed in section 4 is well-suited, since the same deductive rules can be read in both directions. On the other hand, the abstractions provided by deductive rules are rather primitive for integrating schemata. Concept languages (resp., classes) offer a better support through their generalization and aggregation hierachies. We have made a first attempt with query classes to schema integration of distributed relational databases [28]. The system represents the base relations and their abstractions to the integrated schema uniformly as query classes. The deductive interpretation is used for accessing the base relations from queries formulated at the level of the integrated schema. Object-oriented principles come into play when organizing the communication between the external databases: each database can be seen as a class with a small interface of methods offered to the integrated system.

Schema evolution. From the object base perspective, schema descriptions are statements about the world just as their instances are. The prejudice that the schema of a database is not likely to change has led in many systems to a strict separation: schema objects are manipulated at design time only, instance objects at run time. This separation has been recognized as too restrictive for applications where schema knowledge frequently changes, e.g., in design environments. We argue that our object model is particularily well-suited for such situations because its representation does not distiguish instance and class objects. Instead, the relationship between classes and instances is constrained by predefined axioms. If a class is modified then it may violate one or more axioms. Then, instead of rejecting the class modification we can adapt the instances by the view update algorithms outlined in section 4.

6 Conclusions

The main contribution of this paper is an integrated view of queries as classes, rules, and concepts. The idea is to represent queries structurally like concepts and then map them to deductive rules. There is still work to be done on closing the gap between the semantics between concept expressions and deductive databases: the former makes statements about all possible (potentially infinite) models with open world assumption, the latter is fixed to perfect (always finite) models with closed world assumption.

The second contribution is the application of this uniform view to a system architecture consisting of only a few components. View update and maintenance are defined in the deductive interpretation of queries, optimization profits from the object structure and concept hierarchy. A deeper integration of concept and deductive view is still under investigation.

Previous integration efforts concentrated on two of the three aspects of queries. Similar to our approach, [37] integrates concepts with classes but with a fixed and very simple type system (only lattice of attribute names). Queries are mapped to algebraic expressions. Therefore, there is no easy way of parameterizing a query (resp. view definition) like the one offered in our approach. View update is defined for object-preserving operators (subset of our's). Updates on derived attributes are disallowed. The inverse of view update, view maintenance, is not addressed. [1] presents a query language combining deduction with object-orientation. All variables are typed and methods may occur as functional expressions within the rule body. Membership of an instance (a complex value bound to a variable) to a class can be explict or derived by a rule. However, a query itself is not a class. An integration of deductive rules and concept languages is investigated in [17]. The hybrid language $\mathcal{AL}$-log embodies two subsystems for structural knowledge about objects and concepts (based on the concept language $\mathcal{ALC}$) and for relational knowledge between objects (expressed by Datalog rules with variables bound to concepts). These hybrid rules are evaluated by a special constrained SLD-derivation procedure.

Several applications of the implementation of query classes in ConceptBase have validated the usefulness of the idea but also pointed out the need for various extensions, both from the user side (imprecise and intelligent question answering) and from the system side (integration of multiple formalisms). The current implementations of these extensions are mostly of a proof-of-concept nature, their deep and efficient integration into the system remains a major challenge.

Acknowledgements. We are indebted to Matthias Jarke who triggered the idea of this paper and has been a constant source of advice. Many thanks also to our project partners, esp. Werner Nutt, Martin Buchheit and Maurizio Lenzerini, for fruitful discussions.

References

[1] S. Abiteboul, "Towards a deductive object-oriented database language", *Data & Knowledge Engineering*, 5, 1990, pp. 263–287.

[2] A. Borgida, R.J. Brachman, D. McGuiness, and L.A. Resnick, "CLASSIC: A structural data model for Objects", in *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, 1989, pp. 58–67.

[3] F. Bry, H. Decker, and R. Manthey, "A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases", in *Int. Conf. on Extending Database Technology*, 1988, pp. 488–505.

[4] H.W. Beck, S.K. Gala, and S.B. Navathe, "Classification as a query processing technique in the CANDIDE semantic data model", in *Proc. 5th Int. Conf. on Data Engineering*, 1989, pp. 572–581.

[5] M. Buchheit, M.A. Jeusfeld, W. Nutt, and M. Staudt, "Subsumption between queries to object-oriented databases", Report Aachener Informatik-Berichte 93-9, 1993.

[6] R.J. Brachman, D.L. McGuinness, P.F. Patel-Schneider, L.A. Resnick, and A. Borgida, "Living with CLASSIC: When and how to use a KL-ONE-like language", in *Principles of Semantic Networks*(Sowa J.,ed.), Morgan Kaufmann, 1991.

[7] A. Borgida, "From type systems to knowledge representation: natural semantics specifications for description logics", *Int. Journal of Intelligent and Cooperative Information Systems* **1**(1), pp. 93–126, 1992.

[8] C. Beeri and R. Ramakrishnan, "On the power of magic", in *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 1987.

[9] R.J. Brachman and J.G. Schmolze, "An overview of the KL-ONE knowledge representation system", *Cognitive Science* **9**(2), pp. 171–216, 1985.

[10] S. Bergamaschi, C. Sartori, "On taxonomic reasoning in conceptual design", *ACM Transactions on Database Systems* **17**(3), pp. 385–422, 1992.

[11] U.S. Chakravarthy, J. Grant, and J. Minker, "Logic-based approach to semantic query optimization", *ACM Transactions on Database Systems* **15**(2), pp. 162–207, 1990.

[12] S. Ceri, G. Gottlob, and L. Tanca, *Logic programming and databases*, Springer-Verlag, 1990.

[13] P.S. Chen, R. Hennicker, M. Jarke, "On the retrieval of reusable software components", in *Proc. 2nd Int. Workshop on Software Reuse*, Lucca, Italy, 1993.

[14] S. Ceri and J. Widom, "Deriving production rules for incremental view maintenance", in *Proc. 17th Int. Conf. on Very Large Databases*, pp. 577–589, 1991.

[15] U. Dayal, "Active database management systems", in *Proc. 3rd Int. Conf. on Data and Knowledge Bases*, Jerusalem, Israel, 1988.

[16] F.M. Donini, M. Lenzerini, D. Nardi, and W. Nutt, "The complexity of concept languages", in *Proc. 2nd Int. Conf. Principles of Knowledge Representation and Reasoning*, 1991.

[17] F.M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf, "A hybrid system with datalog and concept languages", in *Trends in Artificial Intelligence*, (Ardizzone E., Gaglio S., Sorbello F., eds.), LNAI 549, Springer Verlag, pp. 88-97, 1991.

[18] K. Eshghi and R.A. Kowalski, "Abduction compared with negation by failure", in *Proc. 6th Int. Conf. on Logic Programming*, (Levi G., Martelli M. eds.), Lisbon, MIT Press, pp. 234–254, 1989.

[19] B. Freitag, H. Schütz, and G. Specht, "LOLA–A logic language for deductive databases and its implementation", in *Proc. 2nd Int. Conf. on Database Systems for Advanced Applications*, pp. 216–225, 1991.

[20] R. Fagin, J.D. Ullman, and M.Y. Vardi, "On the semantics of updates in databases", in*Proc. of 2nd ACM SIGACT-SIGMOD*, pp. 352–365, 1983.

[21] P. Gärdenfors, *Knowledge in flux*, MIT Press, 1988.

[22] A. Goldberg and D. Robson, *Smalltalk-80: the language and its implementation*, Addison-Wesley, 1983.

[23] B. Hollunder, W. Nutt, M. Schmidt-Schauss, "Subsumption algorithms for concept description languages", in *Proc. 9th European Conf. on Artificial Intelligence*, pp. 348–353, 1990.

[24] M. Jarke (ed.), *ConceptBase V3.1 user manual*, Report Aachener Informatik-Berichte Nr. 92-17, RWTH Aachen, Germany, 1992.

[25] M. Jeusfeld, *Update control in deductive object bases* (in German). Infix-Verlag, St.Augustin, Germany, 1992.

[26] M. Jeusfeld and M. Jarke, "From relational to object-oriented integrity simplification", in *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases, LNCS 566*, Springer-Verlag, pp. 460–477, 1991.

[27] M. Jeusfeld and M. Staudt, "Query optimization in deductive object bases", in *Query Processing for Advanced Database Applications*, (Freytag et al., eds.), Morgan-Kaufmann, 1993.

[28] A. Klemann, *Schema integration of relational databases* (in German), Diploma thesis, Universität Passau, Germany, 1991.

[29] M. Kifer, W. Kim, Y. Sagiv, "Querying object-oriented databases", in *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, San Diego, Ca., 1992, pp. 393–402.

[30] A.C. Kakas and P. Mancarella, "Database updates through abduction", in *Proc. 16th Int. Conf. on Very Large Databases*, 1990, pp. 650–661.

[31] J. W. Lloyd and R. W. Topor, "Making PROLOG more expressive", in *Journal of Logic Programming*, pp. 225–240, 1984.

[32] J. Mylopoulos, A. Borgida,M. Jarke, and M. Koubarakis, "Telos: a language for representing knowledge about information systems", in *ACM Trans. Information Systems* **8**(4), pp. 325–362, 1990.

[33] B. Mitschang, "Extending the ralational algebra to capture complex objects", in *Proc. 15th Int. Conf. on Very Large Databases*, Amsterdam, 1989, pp. 297–305.

[34] S. Naqvi and T. Tsur, *A logical language for data and knowledge bases*, Computer Science Press, 1989.

[35] A. Olivé, "Integrity constraints checking in deductive databases", in *Proc. 17th Int. Conf. on Very Large Databases*,1991, pp. 513–524.

[36] T. Rose, M. Jarke, J. Mylopoulos, "Organizing software repositories - modeling requirements and implementation experiences", in *Proc. 16th Int. Computer Software & Applications Conf.*, Chicago, Ill., 1992.

[37] M.H. Scholl, C. Laasch, and M. Tresch, "Updatable views in object oriented databases", in *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases*, Munich, Germany, 1991.

[38] M. Staudt, *Query representation and evaluation in deductive object bases* (in German), Diploma thesis, Universität Passau, Germany, 1990.

[39] G. Steinke, "Design aspects of access control in a knowledge base system", in *Computers & Security*, 10, 7, 1991, pp. 612–625.

[40] H.J. Thoennissen, *Design and implementation of an object algebra for a deductive object base system* (in German), Diploma thesis, RWTH Aachen, Germany, 1992.

[41] L. Vieille, P. Bayer, V. Küchenhoff, A. Lefebvre, and R. Manthey, *Documentation for EKS-V1*, Technical Report TR-KB-36, ECRC, 1991.

[42] L. Vieille, "Recursive axioms in deductive databases: The query-subquery approach", In *Proc. 1st Int. Conf. on Expert Database Systems*, 1986.