Authors:	Niemi Timo, Christensen Maria, Järvelin Kalervo
Name of article:	Query language approach based on the deductive object-oriented database paradigm
Year of publication:	2000
Name of journal:	Information and Software Technology
Volume:	42
Number of issue:	11
Pages:	777-792
ISSN:	0950-5849
Discipline:	Natural sciences / Computer and information sciences
Language:	en
School/Other Unit:	School of Information Sciences

URN: http://urn.fi/urn:nbn:uta-3-748
DOI: http://dx.doi.org/10.1016/S0950-5849(00)00118-X

Query language approach based on the deductive object-oriented database paradigm

T. Niemi[1], M. Christensen[1], K. Järvelin[2]

University of Tampere, FIN-33014, Finland
Departments of Computer and Information Sciences[1] and Information Studies[2]

Abstract: The integration of data-oriented (structural), behavioral and deductive aspects is necessary in next generation information systems. Therefore the deductive object-oriented database paradigm offers a very promising starting point for the implementation of these kinds of information systems. So far in the context of this paradigm a big problem has been the lack of a query language suitable to an ordinary end user. Typically in existing proposals for deductive object-oriented databases the user has to master well both logic-based rule formulation and object-oriented programming. In this paper we introduce a set of high-level quering primitives which increases the degree of declarativeness compared to deductive object-oriented query languges proposed so far. In terms of these primitives it is possible to offer for end users such application-specific concepts and structures whose interpretation is obvious to users but whose specification is too demanding for them. By combining these primitives in queries the user can integrate data-oriented, behavioral and deductive aspects with each other in a concept-oriented way. Our query language approach is based on the incorporation of deductive aspects to object-orientation. Among others this means that deductive aspects of objects are inherited in the specialization/generalization hierarchy like any other properties of objects.

Keywords: Deductive object-oriented databases, query languages, query formulation, integration of deductive and object-oriented aspects

1. Introduction

Due to the increasing complexity of information systems both expressive power and modeling capabilities of relational databases have proven insufficient. The database community has tried to solve this problem both by extending the relational model and by developing new data models. For example, transitive relationships among data are very usual in information systems. However relational databases do not support the definition and manipulation of this kind of data because it is impossible to produce the transitive closure of a binary relation with the relational algebra [2]. Therefore the generalized transitive closure operation (see, e.g., [1,7]) has been proposed as an extension to relational algebra. Furthermore, in order to support OLAP ("online analytical processing") applications the relational query language SQL 3 contains as a standard feature the multi-dimensional data grouping and aggregating capability [8].

Deductive databases, object-oriented databases and deductive object-oriented databases (DOOD) are database paradigms which have been proposed as alternatives to relational databases. Deductive databases or logical databases (see e.g. [43]) are databases in which the derived information is defined by logic-based rules (called the IDB or Intensional Data Base) from the existing data (called the EDB or Extensional Data Base). Usually the IDB is represented by recursive Horn clauses. Object-oriented databases have no universally accepted precise model although their features have

been standardized (see [5]). They contain many similar features which are typical of object-oriented analysis and design methods (see e.g. [13,36]). In fact, object-orientation is an interesting combination of data-oriented aspects typical of early semantic data models (see e.g. [12,37,38]) and principles (e.g. encapsulation) found useful in programming. For example, generalization/specialization hierarchies of object-orientation are very similar to IS-A hierarchies of semantic data models. The great potential in object-orientation is due to the fact that both data-oriented (structural) and behavioral aspects are in the same framework.

Deductive object-oriented databases (see e.g. [4]) aim at combining the strengths of deductive databases and object-oriented databases. Both on the basis of deductive and object-oriented databases it is possible to offer the power of a Turing machine in their languages [44]. In other words the combination of these database paradigms does not actually produce new expressive power. The strength of deductive databases is a declarative query language whereas object-oriented databases have a great modeling power among related data.

We agree with those authors (see e.g. [19]) who emphasise that the integration of data-oriented (structural), behavioral and deductive aspects is necessary in next generation information systems. This means that a deductive object-oriented database paradigm offers a very promising starting point for implementation of next generation information systems. However the existing proposals for DOOD query languages require from the user programming skills – often both logic programming and object-oriented programming skills. Unlike the existing proposals for DOOD query languages, we think that the user query language should contain a collection of high-level object-oriented and deductive primitives in terms of which information is represented. From the view point of the user, ad hoc query formulation consists of combining these primitives and of specifying conditions over them. We argue that in next generation information systems it is necessary to hide large and complex deductive specifications from the user by embedding them in such primitives which the user can interpret as obvious application-specific concepts and structures. In object-oriented query languages (see e.g. [6,9]) this approach has already been applied because the user can use methods, which specify the functionalities in an application domain, without knowing their definitions. In this paper we show that the definition of deductive specifications (knowledge derived from the existing information) can also be hidden from the user in terms of our primitives.

The strength of the object-oriented approach is its capability to organize things as they appear in the application domain. Because object-orientation organizes relationships and behavior among related data in an easily interpretable way it is desirable that our language does not violate this feature in any way. Therefore the deductive primitives in our query language approach are incorporated seamlessly to the object-oriented primitives. This is revealed among others so that deductive specifications are inherited in the specialization/generalization hierarchy as any property of a class (object type in our terminology).

The rest of the paper is organized as follows. In Section 2 we introduce different query language approaches proposed for deductive databases, object-oriented databases and deductive object-oriented databases. Special attention is paid to those aspects which make query formulation troublesome from the view point of the ordinary end user. In

Section 3 we introduce those primitives in terms of which data-oriented, behavioral and deductive aspects are represented in our DOOD. In this section we also introduced our sample DOOD described by these primitives. Section 4 introduces constructs of our query language based on primitives in Section 3. In Section 4 we also formulate sample queries in our query language. In Section 5 we discuss the properties and prototype implementation of our language. Summary is given in Section 6.

2. Deductive object-oriented query formulation

Deductive object-oriented query formulation combines aspects related to query formulation both in deductive databases and object-oriented databases. Therefore we consider first query formulation in these database paradigms. Special attention is paid to those factors which are cumbersome for an ordinary end user.

2.1. Query formulation in deductive databases

Typically the derived information or the IDB in deductive databases is defined by Horn clauses from the explicitly represented data which is also defined by Horn clauses. Horn clauses are logical rules that can be non-recursive or recursive. In defining derived information beyond the relational expressive power, the user often has to give recursive rule definitions (e.g. the definition of the transitive closure of a binary relation). Very often a deductive database is defined as a Datalog program which consists of Horn clauses without function symbols, i.e. only variables and constants are allowed in arguments of predicates.

In the mutual comparison of the deductive databases and the object-oriented databases it has often observed (see e.g. [44]) that deductive databases give a better starting point for declarative query languages than object-oriented databases do. However declarativeness in itself is a very relative concept. Declarativeness is a property of a language which allows a person to express the result what (s)he wants rather than how the result is computed. Thus it would be more appropriate to speak about the degree or level of declarativeness. Declarativeness of a language depends on the abstraction level of the primitives of a language. In other words we have to ask whether or not the degree of declarativeness is right in deductive databases from the view point of an ordinary end user. This means that we have to pay attention to what kinds of skills are required from the user in query specification.

There are several aspects which make query formulation in deductive databases troublesome for users. First the user must be able to formulate rules in terms of logic. In addition the user must understand the pattern matching mechanism (unification of terms) within rules and between a query and rules. An ordinary end user is not familiar with this kind of way of thinking. In complex cases rule-based query formulation consists of several definition layers at different abstraction levels which means large specifications. Second, due to the fact that recursive rule formulation is usual in deductive databases the user has to master the recursive way of thinking well. This is a very hard requirement because the user must formulate generally rules for transferring from one level of recursion to another and in addition (s)he must know how recursion is terminated by some exit rule. Among others, a general recursive rule definition presupposes that the user understands how unique variable names are generated for each recursion level and how these variables are instantiated. Third, in

existing deductive database systems the user also has to master the underlying evaluation method on the basis of which rules are processed. In [24] we have demonstrated in the context of the evaluation strategy of Prolog that the user can give a correct specification in the logical sense but computation related to it will never terminate due to the processing strategy. By taking into account the processing strategy the specification can be represented so that computation related to it will terminate. In some existing deductive database systems there are several evaluation techniques. For example, in Coral [34] the user controls the choice of the underlying evaluation techniques. This presupposes that the user masters the available evaluation techniques well.

Also visual query language approaches have been proposed to represent the same information which is specified textually in deductive databases. For example, in VQL [45] a visual query is represented as a set of windows. Each window consists of a header and a body which has a clear analogy to the corresponding parts of a Datalog rule. On the one hand this gives great expressive power for the user and on the other hand the user has to specify queries on the basis of visual constructs of a low abstraction level. One consequence from this is that in complex queries several nested windows are needed. Sometimes this kind of query formulation resembles, from the view point of the user, rather 'visual module design' than a truly declarative specification.

From the above we draw the conclusion that query formulation in deductive databases is too demanding for an end user because he must master logic programming skills. Therefore we proposed in [24-26] that computation of complex transitive relationships among data should be embedded in high-level operations instead of specifying recursive rules. We have shown that this approach facilitates query formulation considerably from the view point of the user. Here we take a step (we believe an important one) forward by proposing that a complex recursive specification is offered for the user as an easily interpretable application-specific concept which can be attached to some object type (class) like any other property.

2.2. Query formulation in object-oriented databases

Currently most object-oriented database systems (OODBS) are based on some imperative object-oriented language. It is obvious that object-oriented languages like C++ or Smalltalk are not more declarative than traditional imperative languages such as Pascal and C. Instead, Voltaire [11] represents a set-oriented, object-based database programming language in which declarative quering primitives are integrated with imperative programming primitives. At the beginning of the 1990's there were several commercial OODBSs but only O_2 [9] offered a real query language intended for end users [6]. It has been widely recognized (see e.g. [42]) that the use of declarative query languages is difficult in the context of object-oriented databases. This is because the object-oriented database paradigm lacks an exact universally accepted model. Although progress has taken place in developing declarative object-oriented query languages they are not yet suitable for ordinary end users.

In order to make possible truly declarative query languages different algebras (see, e.g [3,40,41] have been proposed as the underlying query model of object-oriented databases. Typically these algebras consist of relational operations completed by

some additional operations such as aggregation and data restructuring (nest and unnest) operations. An algebra-based approach to object-oriented query languages means that their expressive power is restricted, i.e., they do not have the computational completeness which is typical of object-oriented programming languages. On the other hand the ordinary end user very seldom needs such a great expressive power.

The starting point in O_2 is to offer two languages: an object-oriented programming language for writing complex applications and a query language for specification of simple information needs [6,9]. The history of the query language of O_2 is given in [6] and now it is called OQL. In addition to ad hoc quering the OQL can be used to write parts of programs which can be integrated flexibly with the code written by any object-oriented programming languguage of O_2. In the integration the impedance mismatching is avoided because programming and query languages have a common type system.

Consider query specification in OQL from the user view point. Although there are several other object-oriented SQL-like query languages with or without prototype implementations we think that OQL gives a good overview on query formulation with an object-oriented query language. Although the user need not to master actual algorithm programming in OQL formulation the user needs to master many things related to object-orientation. For example, the user has to understand that an object of a class (or object type) can be considered as an instance of any of its superclasses. Further (s)he must understand the inheritance mechanism in a class hierarchy in order to know what attributes and operations (or methods) are available for objects belonging to a specific class. In addition, (s)he must understand the notion of object identity because it provides the means to reference objects and thus allows a specific object to be shared by others. In OQL there are two kinds of information available to the user [6]: objects (object types at the schema level) and literals (literal types at the schema level). The user has to distinguish conceptually different kinds of information from each other. For example, objects have an identity whereas literals do not have.

In query formulation based on OQL the user has to combine different iterators (e.g. select-from-where, grouping and sorting iterators) with each other. Unlike in SQL, the user has to master the iterative way of thinking in applying the select-from-where iterator in OQL. Thus the user must understand how a variable is instantiated with different objects until all objects belonging to the class, to which a variable refers, have been dealt. It is also possible that the select-from-where iterator expresses relationships among several variables. In this case the user has to know how variables depend from each other. Furthermore it is possible to nest select-from-where iterators which means that there may be several instantiations of a variable for one instantiation of some other variable. This kind of specification is impossible if the user is not able to think iteratively.

It is typical of OODBSs that more complex objects are constructed by applying available constructors such as set, list, tuple and tree constructors recursively to atomic types such as integer, string, Boolean etc. In OQL like in other object-oriented query languages the user has to master various constructors and their use. It has been realized (see, e.g. [7]) that the use of constructors makes query formulation quite complicated for an ordinary end user. This is especially true in such queries where the

user needs to nest constructors with each other. From the facts introduced above we draw the conclusion that the degree of declarativeness of object-oriented query languages is lower than that of SQL. In this paper we aim at offering such a query language for the user where (s)he need not master the iterative way of thinking. In object-oriented query languages, also in OQL, the attributes and methods are separated clearly from each other, i.e. they are represented in different ways in queries. In our approach the user specifies attributes, methods and deductive aspects related to object types in a very similar way and the user can think that they are some properties related to object types.

2.3. Query formulation in deductive object-oriented databases

There are two main approaches to implement deductive object-oriented databases. One of them is to integrate a logic-based language with an imperative object-oriented programming language (see e.g. [4,39]). Because in this approach the resulting system consists of two separate systems the user has to master well both logic programming and object-oriented programming. In addition, the user must be able to synchronize two kinds of codes with each other. It is obvious that an ordinary end user is not able to make queries in this approach. In the other approach (see e.g. [22,33]) only one language, which contains both logic-based and object-oriented aspects, is offered for users. Usually, the only language of this approach is achieved by extending a deductive database language with object-oriented facilities. On the other hand this makes a deductive database language more capable of organizing complex data with their related behavior but also increases the complexity of the language. Due to the factors mentioned above a pure deductive database language is too complicated for end users. Therefore it is clear that a deductive database language extended with object-oriented facilities is not a suitable query language for end users. In fact we use this kind of language for implementing our query language intended for end users. In this paper we develop a framework which combines the strengths of deductive and object-oriented databases based on high-level primitives. We, as many other authors (see e.g. [32]), think that incorporation of the rich modelling capabilities of object-orientation and declarativeness of deductive databases into the same framework would offer a significant breakthrough.

Also different logics such as F-logic [18] and Transaction F-logic [17] have been developed to support object-oriented aspects in a logical framework. Their purpose is to provide a uniform theoretical foundation for deductive object-oriented databases. However the abstraction level of the notation is the same in these logics as in standard logics, e.g. the recursive way of defining is needed. Although they offer an exact and general starting point for implementing deductive object-oriented databases it is obvious that a query language intended for ordinary end users must be based on primitives on a higher abstraction level. Also other authors have observed that such query formulation, which contains both deductive and object-oriented aspects, is complex for end users. For example, G-log [29] tries to facilitate this kind of query formulation by offering a graph-based way of representing queries. In fact G-log can be seen as a graph-based representation for first-order logic. Here we believe that users need in query formulation primitives on a much higher abstraction level.

2.4 Goals for deductive object-oriented query formulation

Our aim in this paper is to introduce a novel query language approach suitable to end users. It is based on the DOOD paradigm that integrates object-orientation and deduction. The following goals are pursued by our deductive object-oriented query language.

- Query formulation in it should be such that the user need not master ways of thinking typical of programming, such as iterative or recursive thinking. Query formulation should be based truly on declarative specification.

- The query language should be based on such powerful general-purpose constructs in terms of which it is possible to express application-specific concepts and structures in a natural way. It is obvious that in next generation information systems it will be necessary to offer a large set of application specific concepts so that the user need not know how these concepts are produced from existing data. This is due to the fact that in application domains there are a lot of such application-specific concepts and structures whose interpretation is obvious to users but whose specification is too demanding for them. Often in complex concepts several data-oriented (structural), behavioral and deductive aspects have to be integrated with each other.

- The combination of the constructs of the query language with each other should be flexible. This is a necessary feature to guarantee a powerful ad hoc querying capability for the language. By combining application-specific concepts and structures with each other and by specifying conditions among them ad hoc queries are specified.

- The abstraction level of the constructs of the language must be high. For example, it is desirable that the user need not separate data-oriented (structural), behavioral and deductive aspects conceptually from each other. Rather the user should be allowed to think that (s)he has a collection of application-specific properties related to the object types available to him/her.

3. The representation of deductive object-oriented databases

Due to the various ways to implement DOODs there is no established set of primitives of which a DOOD can be seen to consist. Here we make a proposal for those primitives in terms of which data-oriented (structural), behavioral and deductive aspects of a DOOD are represented. Our idea is to hide the definition of deductive aspects from the user according to the same ideology why the definition of behavioral aspects (methods) is hidden from the user in the object-orientation. If the user should specify explicitly deductive aspects as in existing query languages of DOODs then the manipulation of the behavioral and deductive aspects would be based essentially on different abstraction levels. Because the object-orientation is a powerful way of organizing related data with their behavior it would be desirable that the deductive aspects could be embedded in primitives similar to those used in the object-orientation. Otherwise the integration of deductive aspects and object-orientation cannot be seamless. Also in the inheritance mechanism data-oriented, behavioral and deductive aspects should behave in the same way.

3.1 The primitives for deductive object-oriented databases

In our approach the user sees that an application domain consists of objects, relationships among them and properties which can be intrinsic to an object or mutual to several objects. Because in our approach deductive aspects are incorporated to object-orientation we introduce typical object-oriented primitives first briefly. We assume that each *object* is uniquely distinguishable from other objects. In other words we assume that a unique *identity* is assigned to each object and it is used to refer uniquely to the object. An object possesses properties that are intrinsic to it. The characteristics of objects are called *attributes* of an object. An object has also properties that describe functionality related to an object. Each functional property has to be implemented as a piece of code and it is called a *behavior* (or a method) of an object. It is typical of the object-orientation that the implementation details are encapsulated inside the object and the user need not be aware of them.

In the database area two essential abstraction levels or the schema and instance levels are usually distinguished. Roughly speaking the schema level describes how data are organized whereas the instance level contains actual data. Therefore we give for each primitive the terminology that is used on the schema and instance levels. In other words our concept 'object' belongs always to the instance level. Its corresponding concept on the schema level is called *'object type'*. Often in object-oriented databases object types are also called classes.

A *specialization/generalization* hierarchy or an *is-a hierarchy* groups semantically related object types together. In the object-orientation this hierarchy is often called also a class hierarchy. Let X and Y be some object types so that X *is-a* Y holds. On the instance level this means that each object belonging to the object type X belongs also to the object type Y. On the schema level we say that the object type Y is a generalization of the object type X. Very often X is called a *subobject type* of Y and Y is *superobject type* of X. In addition to its specific properties an object type *inherits* the properties of its superobject type(s). On the instance level this means that an object has attribute values for all specific and inherited attributes and it has all specific and inherited behavior. In a specialization/generalization hierarchy there is always an object type which has no superobject type. We call this object type *main object type* because it represents the highest abstraction level in an is-a hierarchy. On the instance level the specialization/generalization hierarchy has the following interpretation. An object belonging to a certain object type belongs also to its all superobject types. If we consider an object as an instance of its superobject type then we can apply only those properties that are available to the superobject type at hand.

In addition to is-a hierarchies we have to express explicitly those mutual relationships among object types which are based on some principle recognized in the application domain. For this purpose we have a primitive called *association type*. It depends on the application domain at hand how many different association types we have. In the representation of an association type we have to specify those object types which participate in the association type and in addition possibly those attributes which describe characteristics of an association type. It is possible that the same object type occurs several times in the association type. In order to distinguish their occurrences from each other a specific *role* is assigned to each occurrence of an object type. Intuitively a role gives the semantic interpretation associated with an object type.

On the instance level it has to be expressed explicitly what objects of the object types participating in an association type are related to each other. A *link* connects individual objects to each other according to the association type so that one object from each object type (role) participating in the association type is in a link. In a link individual objects are expressed by their object identities and in it there is some attribute value for each attribute in the corresponding association type. The collection of all links is called an *association* or it is an instance of the association type. Because an association consists of links of the same structure it means that an association is represented as a relation. This gives a good starting point for the extensive reachability among semantically related data that is a well-known strength of the relational model. Information in our association types is usually represented in object-oriented databases so that an object type (class) has an attribute that contains object identities of another object type whose attribute in turn can contain object identities of some other object type etc. In object-oriented databases object types with these kinds of attributes are often called class-attribute hierarchy (see e.g. [46]). One of the problems related to object-oriented databases is the manipulation of class-attribute hierarchies in some order other than the one in which they were originally established. To our knowledge, there is no such an object-oriented query language that would support the manipulation of class-attribute hierarchies in another order than in the established order. We have chosen such a representation for our association types which does not restrict manipulation to any specific order.

We agree with the opinion of Parsons and Wand ([30-31]) that new actual object types (classes) should be established only in such cases that an object type contains information that cannot be inferred from existing object types. However it is clear that in application domains there is also a need to group objects which share the common selection criteria based on information available to them. The objects satisfying the same selection criteria form a collection of objects that refers to some obvious concept in the application domain. In our approach concepts formed in this principle are called *deductive object types*, i.e. their nature is quite different from the actual object types. In application domains there is often a need to treat deductive object types in their own ways. Therefore it is appropriate to offer deductive object types ready for users, i.e. they do not have to define them. In fact the user need not know whether the object type of interest is an actual object type or a deductive object type.

The definition of a deductive object type is given on the basis of some existing actual object type by specifying conditions among the attributes and methods related to this object type. The underlying actual object type can participate in several association types. Thus, also information in these association types and information reachable via them can be used in the definition of a deductive object type. This starting point affords the possibility of specifying complex selection criteria for the underlying object type. It offers a great derivation power from existing information in respect to semantic data models (see e.g. SDM [12]) where only attributes can be used for definition of derived types.

It is typical that the definition of a deductive object type is attached to some underlying actual object type in the is-a hierarchy at hand. We can apply this definition of a deductive object type in the subhierarchy that starts from this actual object type. This is because in an is-a hierarchy the properties of the underlying object type are included in the properties of its subobject types. If the definition of a

deductive object type is applied in the context of some subobject type it means that the selection criteria of a deductive object type is applied among objects belonging to this subobject type. Thus it is meaningful to speak about inheritance of deductive object types. Inheritance of deductive object types seems a very powerful facility because an is-a hierarchy consists of object types whose nature is similar. Let suppose that we have the *is-a* hierarchy Person - Employee - Teacher and let Sex be one of the attributes of the object type Person (thus it is also via inheritance an attribute for the object types Employee and Teacher). In this case it is possible to define a deductive object type Female in the context of the object type Person which consists of those Person objects who are female (its definition based on the attribute Sex is invisible to the user). Now via inheritance we can apply this deductive object type as such in the context of object types Employee and Teacher which give female employees and teachers, respectively.

Analogously with the notion of the deductive object type we introduce the notion of the *deductive association type*. The association related to some association type represents the links explicitly whereas the links in the association related to a deductive association type must be derived from some existing associations by utilizing available information. Often information needs of users are based on specific structures that are derivable from existing information. Because the deductive association type may contain a very complex derivation rule it is important that this derivation is invisible to the user. It is sufficient that the user is able to interpret the information in the deductive association type. The idea behind the definition of deductive association types is to facilitate query formulation because deductive association types represent inherent and obviously interpretable concept structures in the application domain of interest. Due to the similar nature of actual and deductive association types the user need not know whether actual or deductive association types are being used.

3.2 The sample database

In our example we consider used vehicles and persons that may sell and buy these vehicles. Therefore we can speak about the first, second and so on to the *n*th owner of a vehicle. The example contains information only on those vehicles that have been resold at least once. In Fig. 1 we give a graphical representation on our sample DOOD. The corresponding instance level is represented in Fig. 2 and later on the reader can use this instance level in the evaluation of our sample queries. On the schema level or in Fig.1 object-oriented features are represented like in OML [36] and UML [13]. According to our approach we need new graphical symbols for our deductive components (deductive object and association types). These new symbols have to be connected to the object-oriented symbols so that it is clear what is the basis for their definition. Next we introduce briefly the visual symbols for our modeling primitives.

An object type is visulized as a box divided into three horizontal parts (see Fig. 1). The uppermost part expresses the name of the object type. The attributes of the object type are given in the middle part and its methods are listed in the bottom part. A generalization/specialization hierarchy is represented in terms of double-line arrows. If there is a double-line arrow from an object type A to an object type B it means that B is an immediate superobject type of A. An association type among participating

object types is expressed by connecting each participating object type to a filled circle with a line. A rounded box with two parts is attached to a filled circle for describing information about the association type at hand. In the upper part the name of the association type is given and the lower part lists the attributes of the association type. A possible role name related to a participating object type is expressed in the context of the line.

In our graphical representation the line with a double arrow-head is called a deduction arrow. Deductive object types are described by rectangles with broken borders. A rectangle of this kind contains only the name of the deductive object type. The deductive association type is represented analogously with the association type. However the box is described by broken lines to indicate its deductive nature. A deductive association type is connected to some association type by the deduction arrow. This symbol expresses the association type based on which the deductive association type is defined by using information in it and/or information reachable from it.

End users typically use mainly schema level concepts in query formulation i.e. (s)he must master the symbols depicted in Fig. 1 very well. However the user must understand in our query approach that the variables specified by him/her will be instantiated to the corresponding concepts of the instance level during query evaluation. This means that the user has to have an idea of how object identities are used to connect related data in a DOOD although (s)he need not manipulate them explicitly in query formulation. Typically the same object identities are used in DOOD primitives to preserve semantic relationships among data. In the prototype implementation of our DOOD we have used Prolog++ [22]. In the visualization of our sample DOOD (see Fig. 2) we use object identities generated by Prolog++. In other words they are represented in the form (OT | I) where OT expresses the name of the object type to which an object with the identity I has been established.

Our example contains two generalization/specialization hierarchies with main object types PERSON and VEHICLE and three association types with names SALE, TEACHING and PARENTHOOD. In the association type SALE there are three participating object types: Seller (a role name), Buyer (a role name) and Bought_vehicle (a role name). This association type expresses the sellers and buyers of vehicles i.e., it represents the selling and buying history of used vehicles. In the association type PARENTHOOD we use role names Parent and Child to distinguish two PERSON object types semantically from each other. We believe that the attributes related to our example are mainly self-explanatory. In addition to attributes, the properties of object types may also be methods. The methods *age* (computes the age of a person), and *owned_vehicle* (gives those vehicles a person owns at the moment) are attached to the object type PERSON. The object type VEHICLE has the following methods: *age* (gives the age of a vehicle), *first_sale* (gives the link associated with the first sale case of a vehicle), *last_sale* (gives the link associated with the last sale case of a vehicle) and *number_of_owners* (counts how many owners a vehicle has had so far).

In our sample DOOD there are the following deductive object types: NEW_AUTO_MOBILE (built since 1995), OLD_AUTO_MOBILE (built in 1995 or

before), FEMALE, MALE, YOUNG_PERSON (age less than 25), ADULT (age more than 17) and WELL_EARNING (person's salary is at least 30000). Fig. 1 expresses the object types in whose context they have been defined. Our example contains two deductive association types ANCESTORHOOD (defined based on PARENTHOOD) and OWNING (defined based on SALE). ANCESTORHOOD expresses relationships among ancestors (the role name Ancestor) and their descendants (the role name Descendant). Its attribute Generation expresses the number of the generations between an ancestor and a descendant. OWNING expresses the current owners (the role name Owner) of vehicles (the role name Vehicle). Its attribute Price expresses the price used in the last sale case.

Fig. 1. The schema representation of the sample database.

Fig. 2. The instance level of the sample database

The querying primitives introduced above offer a natural way to represent application-specific concepts and relationships among them. The user needs only to be able to interpret these concepts (without mastering their definition) in a correct way and represent conditions among them. The deductive quering primitives have been designed so that their definitions can hide large and complex definitions. For brevity, our sample deductive object types were quite simple. However, it is worth noting that in the definition of a deductive object type it is also possible to use the information of those association types or information reachable via them, in which the object type used in the definition of a deductive object type participates. For example, we could attach to the object type PERSON the deductive object type BIG_MOTOR_CYCLE_ OWNER, which would define those persons who own at the moment such a motor cycle whose cubic capacity is over 1000. In our approach the user can think in a concept-oriented way i.e. the user can refer to different object types which have conceptually their own meanings. The names of different object types and their properties are like concepts available to him/her. Let us assume that we have the above deductive object type and the user wants to make the following query: 'Give those teachers whose male descendants at the moment own a motor cycle with a large cubic capacity'. In this query the user would combine the following application-specific concepts (the way will be represented in the next Section): *teacher* (an actual object type), *big_motor_cycle_owner* (a deductive object type), *ancestor* (the role name in the deductive association type), *male* (a deductive object type) and *descendant* (the role name in the deductive association type). In this query the user should also refer to *ancestorhood* (a deductive association type) which is an application-specific structure available to him/her.

4. Query language based on DOOD

In our approach we combine the strengths of object-oriented and deductive databases. Therefore information is organized mainly in the object-oriented way whereas query formulation is based mainly on the way typical of deductive databases. We demonstrate that, compared to existing deductive query languages, the degree of declarativeness of a query language can be increased considerably in terms of our DOOD primitives. Next we consider how the user can refer to these quering primitives and combine them.

In our query language, like in deductive database languages, query components separated by commas are interpreted as a logical conjunction. Likewise the notion and notation (starts by a capital letter) of variables are similar to deductive database languages. Among others this means that the same variable in various query components is a shared variable, i.e. a variable of this kind is instantiated to the same value in all query components where it appears. In order to abbreviate our discussion we do not give a complete syntax for our DOOD query language. We think that its essential features can be introduced by giving expressions for our quering primitives.

4.1. Variable expressions

In our query language variables can be associated with objects, links, attribute values or values returned by methods. Objects and links can belong to both actual and deductive object and association types. Next we consider how variables are expressed in the context of different DOOD primitives. One of the ideas of our query language is that similar things are represented in a uniform way. Therefore the user refers to any object of an actual or deductive object type, and to any link of an actual or deductive association type in the same way. Let *abc* denote an (actual or deductive) object type or an (actual or deductive) association type and *X* a variable name selected by the user. Now the expression *abc X* means an object or a link belonging to *abc*, respectively. In other words the user can imagine that the application domain at hand consists of objects of different types and relationships among them which are referable to him/her. Due to the same representation the user need not know which are actual or deductive object/association types. In the context of our sample DOOD the expressions *person X* and *female Y* refer to any person or female object. As explained before objects of object types belong also to their superobject types. Thus the expression *person Y* refers to each object established to the object types *person*, *soldier*, *employee*, *driver*, *teacher* and *student*. Further in our sample DOOD the expression *sale X* means any sale case and in the expression *ancestorhood Y* the variable *Y* refers to any ancestor-descendant relationship. The user need not know how the deductive association type *ancestorhood* has been derived.

In addition to objects and relationships among them the user is interested in properties related to them. For the convenience of the user it is desirable that the same way is used to refer to the properties of both objects and links. The properties attached to objects are attributes and methods whereas the properties related to links are attributes and role names of participating objects. Let *X* be a variable that is instantiated either to an object or a link and *abc* be a property then the notation *X:abc(Y)* means that *Y* is the value of the property *abc* in an instance to which *X* has been instantiated. Thus in our sample DOOD the sequence *male X, X:age(A), X:name(N)* is a query which gives men (the instantiations of X) with their ages (the instantiations of A) and names (the instantiations of N). Analogously the sequence *sale X, X:bought_vehicle(Y), Y:make(M), X:price(Z)* is a query which gives sale cases (the instantiations of *X*), vehicles that were resold in sale cases (the instantiations of *Y*), their makes (the instantiations of *M*) and prices used in sale cases (the instantiations of *Z*). In our language it is also possible to refer directly to a property of a property, i.e. in the above sequence the query components *X:bought_vehicle(Y), Y:make(M*) could be replaced by the query component *X:bought_vehicle:make(M).*

4.2. Expressions for excluding objects belonging to a specific object type

Sometimes the user wants to manipulate objects belonging to a specific object type except for those that belong to its some subobject type(s). Our query language contains an expression in terms of which the user can exclude objects belonging to a specific object type from manipulation. In other words the user is not interested in these objects. The expression of form *neg abc X* means that *X* can be instantiated to other objects in the DOOD except for those which belong to the object type *abc*. Thus in our sample DOOD the expression *neg truck X* refers to all other objects (both vehicles and persons) except for those that belong to the object type *truck*. Further, the sequence *vehicle X, neg truck X* means all vehicles (but not persons) that are not trucks. *Analogously* we can also exclude objects belonging to a deductive object type. For example, if we want to find out those teachers which do not earn well we can do it by the query *teacher X, neg well_earning X*.

4.3. Concatenation expression for object types

In a DOOD we have several actual and deductive object types. By combining them with each other the user can easily refer to conceptually new object sets and specify conditions among them. The combination should be such that it would express the object set of interest in an easily interpretable way. It is important that the user can use in the combination also the exclusion notation neg introduced above. In order to allow the possibility for compact query formulation several query components with the same variable can be represented by one query component as follows. The query components ot_1 *X*, … ,ot_n *X*, where ot_1 … ot_n are some object types in a DOOD, can be represented as the query component ot_1 … ot_n *X*. We call this the concatenation expression of object types. From the above it should be clear that object types in a concatenation expression are associated with each other by the logical conjunction. For example, if the user wants to know those male teachers who earn well (s)he can do it either by the query *teacher X*, *male X, well_earning X* or by applying the concatenation expression *teacher male well_earning X*. In this concatenation expression there are one actual (*teacher*) and two deductive (*male* and *well_earning*) object types. The order of object types in the concatenation expression is free. This means that the expression *teacher male well_earning X* could also be represented e.g. in the order *well_earning male teacher X*. It is worth noting that the excluding notation *neg* applies only to the immediately following object type. If in our sample DOOD the user wants to know girls which are not yet adults (s)he can do it by the expression *neg adult female X*. According to object-orientation an object type can contain objects which do not belong to any of its subobject types. In our query language we can refer to these objects easily by applying the excluding notation *neg* in the concatenation expression. In our sample DOOD the expression *neg driver neg teacher employee X* gives those employees whose profession is neither driver nor teacher.

4.4. Sample queries based on the instantiations of all variables

In the results of queries the user is usually interested only in some properties related to objects or links - not the instantiations of all variables in queries. For example, object identities as instantiations of variables are seldom interesting from the perspective of the user. However we first consider sample queries based on the instantiations of all variables. This is because on the basis of these instantiations it is

possible to demonstrate how related data in query components are connected. By specifying typical comparison operators such as >, <, = etc. in queries it is possible to compare values of properties with each other and constants. Our sample queries are based on our sample DOOD in Fig. 2. In order to save space we give answers only for some queries. The reader can evaluate answers for all queries based on the information given in Fig. 2.

Sample Query 1: Find out those persons whose age is over 35 and who own a vehicle whose age is more than 10.
Expression for Sample Query 1:
person X, X:age > 35, X:owned_vehicle:age > 10.
In this query the method *owned_vehicle* (a property of the person object type) yields a vehicle whose method (a property of the vehicle object type) in turn is *age* (differs from the method *age* applied in the context *X*). Thus we can refer directly to a property of a property in the query component *X:owned_vehicle:age > 10*. The answers X = (employee|347559) and X = (teacher|354860) are given for the result.

Sample Query 2: Give those female employees who earn well and own at this moment a new car whose top speed is more than 190.
Expression for Sample Query 2:
well_earning female employee X, new_auto_mobile car Y, owning Z, Z:owner(X), Z:vehicle(Y), Y:top_speed > 190.
Here we have two concatenation expressions among objects. One consists of one actual (employee) and two deductive (*well_eraning* and *female*) object types and the other of one actual (*car*) and one deductive (*new_auto_mobile*) object types. In addition the query contains one deductive association type (*owning*) and a reference to a property (*top_speed*) of its property (*vehicle*).

Sample Query 3: Select those sale cases where the seller is an ancestor for the buyer.
Expression for Sample Query 3:
sale X, ancestorhood Y, X:buyer(B), X: seller(S), Y:ancestor(S), Y:descendant(B).
This query demonstrates how relationships among related data in two association types (one is actual and the other is deductive) are specified by shared variables.

4.5. Queries based on specifying the result relation

As our previous queries show they contain many such instantiations of variables that actually are not interesting for the user, e.g. they are used to connect related data. Therefore our query language allows the user to describe the result of a query so that it contains only information relevant to the user. A relation is a natural way to represent the result of a query. This has been recognized in the context of relational databases where relations are visualized as easily interpretable tabular representations. Also in our query language the user can specify the form of the result. For this our query language contains the expression
rr(c1, ..., cn) provided q, where
rr(c1, ..., cn) is the form of the result relation specified by the user
provided is a reserved word and
q is any query introduced above.
In the form of the result relation specified by the user *rr* is a result relation name given by the user and *c1, ... , cn* are the attributes of the result relation. The attributes

of the result relation are properties related to DOOD primitives and they are represented in a way introduced above.

Sample Query 4: Give those persons who own a red car currently. The user wants that the result relation contains the names and addresses of persons and the makes and register numbers of cars.
Expression for Sample Query 4:
red_car_owners(O:name,O:address,C:make,C:register) provided car C, owning Y, Y:vehicle(C), Y:owner(O), C:color = red.
This query shows that the result relation can contain data belonging to different object types. The result relation is given in Fig. 3.

Fig. 3. The result relation for Sample Query 4.

Sample Query 5: Now we consider the vehicle with the register number 'LUU-5'. Give its make and the prices used in its first and last sale.
Expression for Sample Query 5:
first_last_prices(X:make, X:first_sale:price, X:last_sale:price) provided vehicle X, X:register = 'LUU-5'.
The query demonstrates that in the description of the result relation we can refer to a property of a property in the same way as anywhere in the query. For example, *first_sale* is a method (a property) attached to object type *vehicle*. It returns a link whose attribute (a property) *price* is.

Sample Query 6: Let us assume that the user is interested in those cars which are owned by male students which a female teacher teaches. In the result the user wants to know the names of teachers and students and the register numbers of cars.
Expression for Sample Query 6:
teacher_student_car(T:name, S:name, C:register) provided female teacher T, male student S, car C, teaching Y, owning O, Y:teacher(T), Y:student(S), O:owner(S), O:vehicle(C).
This query shows that also complex queries can be formulated in a compact and very declarative manner. The result relation contains data from three different object types, it deals with two deductive (*female teacher*, *male student*) and one actual (*car*) object types and one actual (*teaching*) and one deductive (*owning*) association type.

4.6. Queries containing aggregation information in the result relation

Roughly speaking aggregation information is summary data which is computed on the basis of some data set. In our query language a variable has several instantiations. These instantiations form a natural data set among which aggregation information is computed. The aggregation information in the result relation is described as follows *aggr_type(expr)* where *aggr_type* is an aggregation type and *expr* is an expression that produces the data set among which aggregation information is computed. The currently available aggregation types are count, sum, average, minimum, maximum, list denoted by *count*, *sum*, *avg*, *min*, *max* and *list*, respectively. In the result relation there may be several instances of some object type (say B) for one instance of some other object type (say A). Usually this information has been split into several rows in the result relation. If the user wants (s)he can collect this information into one row so

that information related to B is specified as a list in the description of the result relation.

Sample Query 7: Consider vehicles and their owners at the moment. Give the youngest and eldest ages related to both owners and vehicles. Likewise the average ages of owners and vehicles must be computed.
Expression for Sample Query 7: owner_vehicle_analysis(min(O:age), max(O:age), avg(O:age), min(V:age), max(V:age), avg(V:age)) provided owning X, X:owner(O), X:vehicle(V).
The result relation contains several aggregation types related to two object types: *person* (via the role name *owner*) and *vehicle*. The instantiations for objects belonging to these object types are taken from the deductive association type *owning*. The actual aggregated values are computed on the basis of values yielded by two separate *age*-methods. The result of this query is in Fig. 7

Fig. 4. The result relation for Sample Query 7.

Sample Query 8: Give the names and addresses of those men who own one or more automobiles. In addition, the result relation has to contain the makes of automobiles owned by men.
Expression for Sample Query 8: male_auto_owners(X:name,X:address,list(Z:make)) provided male X, auto_mobile Z, owning Y, Y:owner(X), Y:vehicle(Z).
In the deductive association type *owning* there may be several automobiles for one male owner. It means that by applying the aggregation type *list* we are able to collect these automobiles together. In fact the result relation (see Fig. 5) is not anymore a relation in the first normal form but it is the so-called NF^2 (non-first normal form) or nested relation (see e.g. [35]) because the values of the attribute **make_list** are not atomic in the result. The use of our list constructor is similar to list and set constructors of several existing deductive database languages (see e.g. [21]). Our idea is to extend our language in the future so that the general nested aggregation, i.e. the aggregation of aggregated data, is possible.

Fig. 5. The result relation for Sample Query 8.

5. Discussion

Increasing attention has been paid to the possibility of using application-specific concepts to facilitate query formulation. For example, for case-based reasoning systems a concept-oriented data model has been developed by extending the conventional object-oriented data model [10]. It emphasizes the need both for producing derived information (concepts inferable from the explicitly stored data) and for representing contextual aspects on data. Here our attempt has been to find such a *general* set of primitives in terms of which it is possible to represent application-specific concepts and relationships among them. The primitives of our query language are capable of representing data-oriented (structural), behavioral and deductive aspects of application domains. In our approach the definition of different concepts by our primitives is completely invisible to the user (the encapsulation principle). Thus our approach can be characterized as a concept-oriented framework to implement a DOOD query language where the concepts have been predefined for the users.

The organization of our quering primitives is based mainly on object-orientation because it offers, from the view point of the user, an easily interpretable way to understand the relationships among application-specific concepts. This means among others that the selection criteria, on the basis of which a deductive object type is defined, are inherited like any property in the specialization/generalization hierarchy at hand. By applying the definition of a deductive object type in the context of different object types the user is able to refer very intuitively to new object sets which conceptually have their own meaning in the application domain of interest. It is important to note that the selection criteria used in the definition of deductive object types can be very complex, e.g. they may contain recursive definitions. From the view point of the user, it is sufficient that he is able to interpret the meaning of a deductive object type correctly in the application domain at hand without mastering its detailed definition. This is a powerful mechanism to offer querying primitives at a high abstraction level for the user. In our sample queries we demonstrated that, in terms of the concatenation expression, it is possible to form conceptually new selective object sets by combining several deductive and actual object types with each other. The features introduced above increase the degree of declarativeness compared to DOOD query languges proposed so far.

In our query language approach we combine the strengths of object-oriented and deductive databases. Therefore query formulation is based mainly on the way typical of deductive databases because they support a declarative approach to query formulation. For example, by using shared variables in query components similarly as in deductive query languages the user can express semantically related information intuitively and compactly. Our quering primitives have been chosen so that application-specific concepts can be represented and integrated with each other in an intuitive way in terms of them. In the conventional DOOD query languages the user is responsible for the definition of these kinds of concepts. We have analyzed those factors which make their definition too demanding for ordinary end users. It is obvious that in next generation information systems it will be necessary to embed complex (often deductive) and large specifications in application-specific structures and concepts which the user can use as such in query formulation.

In our query language approach the user can think that the application domain of interest consists of objects, relationships among them and the properties related both to objects and relationships. From the view point of the user, query formulation mainly is specification of conditions among these quering primitives. Our quering primitives also represent data sets among which the computation of aggregation information is natural. In order to support data aggregation in the context of our primitives the query language contains typical aggregation operations.

Due to the high abstraction level of quering primitives we have succeeded in removing many immaterial details from query formulation. For example, the user need not know whether he refers to an actual or deductive object/association type. Likewise the user need not separate properties of different kinds because the same way is used to express the data-oriented and behavioral properties of objects. In order to support a uniform query formulation only one syntactical notation is used to refer to things whose nature is similar. For example, in our language the same way is used to refer to the properties of object and association types.

This paper is related to our long-term research project concerning query languages. Our purpose is to find those factors that facilitate query formulation considerably from the view point of the user. We have developed query languages and techniques for their implementation in terms of which specification related to navigation [14-15] and restructuring [27-28] among data can be removed from the user. Likewise we have shown that in deductive databases complex recursive rule definitions for transitive processing can be replaced by a set of operations intuitive to the user [24-25]. In [16] we introduce a language in which the user may specify in a compact and truly declarative way queries which contain complex restructuring, aggregation and transitive processing among data at the same time. We believe that we take in this paper a significant step forward by proposing that specifications are embedded in concepts that can be represented by DOOD primitives introduced in this paper.

Often (see e.g. [20]) one of the strengths of deductive databases is considered that they extend the expressive power of traditional databases by allowing recursion. As we discussed above recursion is too complex a notion for an ordinary end user. To our mind recursive rule-based specification should be removed totally from the user if we want to offer a user-friendly query language. Here we show that recursive specifications can be embedded in deductive association types in a natural way. In our sample DOOD *ancestorhood* was a deductive association type of this kind. To its participating object types the user referred simply by the concepts *ancestor* and *descendan*t, i.e. from the view point of the user they were properties of *ancestorhood*. In the same way recursive specifications can be embedded in methods of objects.

It is obvious that the primitives of our DOOD approach have the computationally complete expressive power. If this expressive power would want to be offered for the user it would mean that the user should have programming skill in order to implement these primitives. We believe that the necessary expressive power for the user can be produced by defining beforehand available concepts (expressed by DOOD primitives) which are combined by our query language. It is important that we can test the appropriateness and adequacy of application-specific concepts before expensive implementation. For this purpose we have developed a deductive object-oriented modeling method, called DOOM [23]. In it our primitives are represented on the basis of set theory and sample queries can be formulated with set theory in a natural way.

We have a prototype implementation for our DOOD query language. For our implementation language we selected Prolog ++ [22]. For the reasons mentioned above Prolog ++ as such is not a suitable query language for ordinary end users. However it is ideal in many respects as an implementation language for DOOD query languages. This is because Prolog++ combines logic programming and object-oriented programming to one homogenous deductive object-oriented programming framework. It supports both typical object-oriented features such as complex objects, object identities, encapsulaition, classes, inheritance, method overriding, late binding and on the other hand logic-based rule formulation typical of deduction. In fact Prolog++ has been implemented on top of Prolog which means that all predicates of Prolog are also available in Prolog ++. In the implementation of a specialization/generalization hierarchy we used mainly the inheritance mechanism of Prolog++. However in Prolog++ an object belongs automatically only to the class to which it has been established. According to our modeling principle an object type contains also those objects which belong to its subobject types. In prototype

implementation it meant that this Prolog++ feature had to be redefined to correspond to our modeling principle.

6. Conclusions

In this paper we propose a novel approach to a deductive object-oriented query language. Our approach is based on explicitly defined primitives in terms of which it is possible to model advanced application domains in a concept-oriented way. This is because different kinds of concepts can be represented with our primitives. The primitives of the approach take into account data-oriented, behavioral and deductive aspects among data. In other words, it integrates essential features of next generation information systems into a uniform framework. In terms of the primitives actual (possibly large and complex) specifications can be hidden from the user. From the view point of the user ad hoc query formulation seems to be a combination of different concepts and expression of conditions among them. Powerful primitives and their free combination offer the necessary expressive power for the user. We gave several sample queries to demonstrate how our quering primitives increase the degree of declarativeness compared to existing DOOD query languages.

References

[1] Agrawal, R, Alpha: an extension of relational algebra to express a class of recursive queries, in: Proc. of the 3rd IEEE Conference on Data Engineering, Los Angeles, Feb. 3-5, (1987) 580-590.

[2] Aho A and Ullman J D, Universality of data retrieval language, in Proc. of the 6th ACM Symposium on Principles of Programming Languages, Texas, (1979) 110-120.

[3] Alhajj R and Arkun M, A query model for object-oriented databases, in: Proc. of the 9th IEEE Conference on Data Engineering, Vienna, April 19-23, (1993) 163-172.

[4] Barja M, Paton N, Fernandes A, Williams M and Dinn A, An effective deductive object-oriented database through language integration, in: Proc. of the 20th VLDB Conference, Santiago, Sept. 12-15, (1994) 463-474.

[5] Cattell R and Barry D (Eds), The object data standard: ODMG 3.0, (Academic Press, San Diego, 2000).

[6] Cluet S, Designing OQL: Allowing Objects to be Queried, Information Systems 23(5) (1998) 279-305.

[7] Dar S and Agrawal R, Extending SQL with Generalized Transitive Closure, IEEE Trans. Knowledge and Data Engineering 5(5) (1993) 799 – 812.

[8] Date, C, An introduction to database systems (7 th edition) (Addison Wesley Longman, 2000).

[9] Deux O, The Story of O2, IEEE Trans. Knowledge and Data Engineering 2(1) (1990) 91 – 108.

[10] Dubizky W, Buchner A, Hughes J and Bell D, Towards Concept-oriented Databases, Data & Knowledge Engineering 30 (1999) 23-55.

[11] Gala S, Navathe S and Bermudez M, Voltaire: A database programming language with a single execution model for evaluation queries, constraints and functions, in: Proc. of the 9th IEEE Conference on Data Engineering, Vienna, April 19-23, (1993) 283-292.

[12] Hammer M and McLeod D, Database Description with SDM: a Semantic Database Model, ACM Trans. Database Syst. 6(3) (1981) 351 - 386.

[13] Harmon P and Watson M, Understanding UML: the developers' quide, (Morgan Kaufmann, San Francisco, 1997).

[14] Järvelin K and Niemi T, An Entity-based Approach to Query Processing in Relational Databases Part I: Entity Type Representation, Data & Knowledge Engineering, 10 (1993) 117-150.

[15] Järvelin K and Niemi T, An Entity-based Approach to Query Processing in Relational Databases Part II: Entity Query Construction and Updating, Data & Knowledge Engineering 10 (1993) 151-186.

[16] Järvelin K and Niemi T, Integration of Complex Objects and Transitive Relationships for Information Retrieval, Information Processing & Management 35 (1999) 655-678.

[17] Kifer M, Deductive and object data languages: a quest for integration, in: Proc. of the 4th Int. Conf on Deductive Object-oriented Databases, Singapore, Dec. 4-7, (1995) 187-212.

[18] Kifer M, Lausen G and Wu J, Logical Foundations of Object-oriented and Frame-based Languages, Journal of ACM 42(4) (1995) 741-843.

[19] Li Q and Lochovsky F, ADOME: An Advanced Object Modeling Environment, IEEE Trans. Knowledge and Data Engineering 10(2) (1998) 255 – 275.

[20] Liu M, ROL: a Deductive Object Base Language, Information Systems, 21(5) (1996) 431-457.

[21] Liu M, Deductive Database Languages: Problems and Solutions, ACM Computing Surveys 31(1) (1999) 27-62.

[22] Moss C, Prolog ++ the power of object-oriented and logic programming, (Addison –Wesley, Cambridge, 1994).

[23] Niemi T, Junkkari M and Järvelin K, Deductive object-oriented approach to systems analysis and its representation with the set theory, Report A-1998-15, Dept. of Computer Science, University of Tampere, 1998.

[24] Niemi T and Järvelin K, Operation-oriented Query Language Approach for Recursive Queries - Part 1: Functional Definition, Information Systems 17(1) (1992) 49-75.

[25] Niemi T and Järvelin K, Operation-oriented Query Language Approach for Recursive Queries - Part 2: Prototype Implementation and Its Integration with Relational Databases, Information Systems, 17(1) (1992) 77-106.

[26] Niemi T and Järvelin K, Advanced Query Formulation in Deductive Databases, Information Processing & Management 28(2) (1992) 181-199.

[27] Niemi T and Järvelin K, A Straightforward Nf2 Relational Interface with Applications in Information Retrieval, Information Processing & Management 31(2) (1995) 215-231.

[28] Niemi T and Järvelin K, The Processing Strategy for the Nf2 Relational Frc-interface, Information and Software Technology 38 (1996) 11-24.

[29] Paredaens J, Peelman P and Tanca L, G-log: a Graph-based Query Language, IEEE Trans. Knowledge and Data Engineering 7(3) (1995) 436 – 453.

[30] Parsons J and Wand Y, Choosing Classes in Conceptual Modeling, Communications of the ACM 40(6) (1997) 63-69.

[31] Parsons J and Wand Y, Using Objects for Systems Analysis, Communications of the ACM 40(12) (1997) 104-110.

[32] Paton N, Cooper R, Williams H and Trinder P, Database programming languages, (Prentice Hall, Cambridge, 1996).

[33] Paton N W, Leishman S, Embury S M and Gray P M D, On Using Prolog to Implement Object-oriented Databases, Information and Software Technology 35(5) (1993) 301-311.

[34] Ramakrishnan R, Srivastava D and Sheshadri P, Implementation of the CORAL deductive database system, in: Proc. of the ACM Sigmod Conference, Whasington, May 26-28, (1993) 167-176.

[35] Roth M A, Korth H F and Silberschatz A, Extended Algebra and Calculus for Nested Relational Databases, ACM Trans. Database Syst. 13(4) (1988) 389-417.

[36] Rumbaugh J, Blaha M, Premerlani W, Eddy F and Lorensen W, Object-oriented modeling and design, (Prentice Hall, 1991).

[37] Shipman D, The Functional Data Model and the Data Language DAPLEX, ACM Trans. Database Syst. 6(1) (1981) 140 – 173.

[38] Smith J and Smith D, Database Abstractions: Aggregation and Generalization, ACM Trans. Database Syst. 2(2) (1977) 105 – 133.

[39] Srivastava D, Ramakrishnan R, Seshadri P and Sudarshan S, Coral++: Adding object-orientation to a logic database language, in: Proc. of the 19th VLDB Conference, Dublin, Aug. 24-27, (1993) 158-170.

[40] Steenhagen H-J, Apens P, Blanken H and de By R A, From nested-loop to join queries in oodb, in: Proc. of the 20th VLDB Conference, Santiago, Sept. 12-15 (1994) 618-629.

[41] Straube D and Özsu M, Queries and Query Processing in Object-oriented Database Systems, ACM Trans. Information Systems 8(4) (1990) 387-430.

[42] Ullman J, Database theory - past and future in: Proc. of the 6th ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems, San Diego, March 23-25, (1987) 1-10.

[43] Ullman J, Principles of database and knowledge base systems, Vol. 1, (Computer Science Press, Rockville, MD, 1988).

[44] Ullman J, A comparison between deductive and object-oriented database systems, in: Proc. of the 2nd Int. Conf. on Deductive Object-oriented Databases, Munich, Dec. 16-18, (1991) 263-277.

[45] Vadaparty K, Aslandogan Y and Ozsoyogly G, Towards a unified visual database access, in: Proc. of the ACM Sigmod Conference, Whasington, May 26-28, (1993) 357-366.

[46] Yong H, Lee S and Kim H-J, Applying signatures for forward traversal query processing in object-oriented databases, in: Proc. of the 10th IEEE Conference on Data Engineering, Houston, Feb. 14-18 (1994) 518-525.