A SQL-like Query Calculus for Object-Oriented Database Systems*

Rudolf Herzig and Martin Gogolla

TU Braunschweig, Informatik, Abt. Datenbanken
Postfach 3329, D-38023 Braunschweig, Germany
e-mail: {herzig|gogolla}@idb.cs.tu-bs.de

Abstract. Currently much effort is being spent on providing object-oriented databases with ad hoc query facilities. In this paper we present a SQL-like query calculus whose major contribution lies in its inherent orthogonality and rigorous mathematical foundation. The calculus is essentially a calculus of complex values but it is defined independently of any concrete database model. The calculus can be used to formulate queries in value-based and object-based data models. Moreover it provides a general facility for the manipulation of complex values.

1 Introduction

Object-oriented database systems (OODB) are usually embedded in a programming language environment providing full computational power. Consequently, designers of OODB at first did not feel the need to integrate ad hoc query facilities into their systems. However, it has been recognized that some associative retrieval is of importance even for OODB [8, 12, 13].

A well-tried ad hoc query language is SQL. SQL came up with relational database systems so that many people regard SQL as a typical relational query language. On the other hand SQL has been successfully adapted to semantic data models [32, 30], and there have been many proposals for applying SQL to OODB [8, 16, 19, 22, 34, 14]. A number of these SQL variants have already found their way into commercial products like OSQL [11] in IRIS, Object SQL [26] in ONTOS, RELOOP [16] and its successor O_2Query [9] in O_2, CQL++ [19] in Ode, XSQL [34] in ORION. A derived form of O_2Query is being considered as the query language OQL of the ODMG-93 standards proposal [15, 35].

Unfortunately the syntax of many of these query languages is illustrated by some examples only, and their semantics is mostly described in an informal way. In other cases the semantics is defined by an involved translation into an *algebra-based* framework (e.g., RELOOP [16]) or a *rule-based* (logic programming oriented) formalism (e.g., ESQL2 [22], XSQL [34]). However with regard to SQL

* in: Proc. Int. Symp. on Object-Oriented Methodologies and Systems (ISOOMS), Palermo, 1994, (E. Bertino and S. Urban, eds.), Springer, Berlin, LNCS 858, pp. 20–39

Work reported here has been partially supported by the CEC under Grant No. 6112 (COMPASS) and BMFT under Grant No. 01 IS 203 D (KorSo).

calculus-based formalisms seem to be more helpful because of their closeness to the basic select-from-where construct found in SQL [42].

In this paper we present a SQL-like query calculus the aim of which is to provide a general framework for SQL-like query languages. Therefore the calculus is defined independently from any concrete database model. In comparison with other query language proposals we stress the following advantages.

Completely orthogonal SQL: In contrast to many approaches which allow for nested queries only in the *from* and *where* clause of a select term our query calculus allows for arbitrary terms, in particular further select terms, to be used in the *select* clause of a select term.

Abstract syntax: The syntax of the query calculus is precisely defined by mathematical notions being independent from a concrete language and thereby covering both context-free and context-sensitive rules in a formal and compact manner.

Straightforward semantics: Each syntactical category is directly given a rigorous formal semantics based on sets.

The query calculus is essentially a calculus of complex values. With that it stands in the tradition of the query calculi proposed for complex object models [37, 10, 43, 2]. However most of these calculi preserve the traditional hierarchical structure of predicate calculus in that terms are used to build formulas but formulas do not contribute to the construction of terms. These frameworks fail to reflect the fact that SQL queries can be nested. Therefore we propose a query calculus with *non-hierarchical* structure, i.e., terms may be built on the basis of formulas.

A concrete SQL-like language proposal which allows for nested queries in the *select*, *from*, and *where* clause of a select term is RELOOP (or $\mathrm{O_2}$Query resp.). It is defined in context of an object model with complex values [39]. However we are not aware of a complete formal description of this language.

The paper is organized as follows. In Section 2 we introduce a general model of complex values. In Section 3 we define a calculus for complex values. Applications of this calculus are sketched in Section 4. We give some concluding remarks in Section 5.

2 Complex Values

Notation: Let sets $\mathrm{S}, \mathrm{S}_1, \ldots, \mathrm{S}_n$ be given. Then $\boldsymbol{\mathcal{F}}(\mathbf{S})$ denotes the restriction of the power set of S to finite sets, $\mathbf{S}^*$ the set of finite lists over S, $\boldsymbol{\mathcal{B}}(\mathbf{S})$ the set of finite multi-sets (bags) over S, and $\mathbf{S_1} \times \ldots \times \mathbf{S}_n$ the Cartesian product of the sets $\mathrm{S}_1,\ldots,\mathrm{S}_n$. Finite sets are written as $\{\mathbf{c_1}, \ldots, \mathbf{c}_n\}$, lists as $\langle \mathbf{c_1}, \ldots, \mathbf{c}_n \rangle$, bags as $\{\!\{ \mathbf{c_1}, \ldots, \mathbf{c}_n \}\!\}$, and elements of the Cartesian product as $(\mathbf{c_1}, \ldots, \mathbf{c}_n)$.

2.1 Atomic Sorts

We define values as instances of data sorts. Data sorts are usually accompanied by specific operations and predicates. All these are summarized in a data signature.

Definition 1 (data signature). A data signature is given by a triple $\Sigma = (S, \Omega, \Pi)$ in which S denotes a set of sorts, $\Omega = \{\Omega_s\}_{s \in S^* \times S}$ a family of $S^* \times S$-indexed operation symbols and $\Pi = \{\Pi_s\}_{s \in S^*}$ a family of S^*-indexed predicate symbols. $\square$

$\omega_{s_1 \dots s_n, s} \in \Omega$ is also written as $\omega : s_1 \times \dots \times s_n \rightarrow s$ and $\pi_{s_1 \dots s_n} \in \Pi$ as $\pi : s_1 \times \dots \times s_n$.

Examples.

1. The following sorts and associated operations and predicates are standard.

   $S \supseteq \{$ nat, int, real, bool, char, string, $\dots\}$
   $\Omega \supseteq \{+, -, *, \mathtt{DIV}, \mathtt{MOD} : \text{int} \times \text{int} \rightarrow \text{int}, \dots\}$
   $\Pi \supseteq \{<, \leq, =, \geq, >: \text{int} \times \text{int}, \dots, \}$

   Data constants are considered as nullary operations symbols, e.g., $42 :\rightarrow \text{int}$ or true $:\rightarrow$ bool.
2. A data signature may also contain *non-standard* data types.

   $S \supseteq \{$ stack, $\dots\}$
   $\Omega \supseteq \{$ empty $:\rightarrow$ stack, push : stack $\times$ int $\rightarrow$ stack,
   pop : stack $\rightarrow$ stack, top : stack $\rightarrow$ int, $\dots\}$

For a data signature Σ we assume an interpretation structure $I(\Sigma)$ to be given.

Definition 2 (interpretation of a data signature). For a given data signature Σ an interpretation structure is defined as a triple $I(\Sigma) = (I(S), I(\Omega), I(\Pi))$.

- $I(S)$ associates each sort $s \in S$ with a set $I(s)$ such that $\perp_s \in I(s)$.
- $I(\Omega)$ associates each operation symbol $\omega : s_1 \times \dots \times s_n \rightarrow s \in \Omega$ with a total function $I(\omega) : I(s_1) \times \dots \times I(s_n) \rightarrow I(s)$.
- $I(\Pi)$ associates each predicate symbol $\pi : s_1 \times \dots \times s_n \in \Pi$ with a relation $I(\pi) \subseteq I(s_1) \times \dots \times I(s_n)$. $\square$

An interpretation structure can be viewed as a many-sorted algebra.

Example. The interpretation of standard data sorts may be fixed as follows.

$I(\text{nat}) = \mathbb{N}_0 \cup \{\perp_{\text{nat}}\}$, $I(\text{int}) = \mathbb{Z} \cup \{\perp_{\text{int}}\}$, $I(\text{real}) = \mathbb{Q} \cup \{\perp_{\text{real}}\}$,
$I(\text{bool}) = \{\mathit{true}, \mathit{false}\} \cup \{\perp_{\text{bool}}\}$,
$I(\text{char}) = A \cup \{\perp_{\text{char}}\}$, $I(\text{string}) = A^* \cup \{\perp_{\text{string}}\}$,

in which A denotes a finite set of characters. Standard operations and predicates have the usual meaning.

An element $\underline{s} \in I(s)$ is called a *value*. Values are written *italic*. For example the constant $42 :\rightarrow$ int generates the value *42*. For each sort s there is a special *null* (or error) *value* $\perp_s$. For example we have $I(\mathtt{DIV})(\mathit{1}, \mathit{0}) = \perp_{\text{int}}$. Null values are generated by $\mathtt{ERROR}_s :\rightarrow s \in \Omega$. For simplicity the index of the null value is dropped in the following.

2.2 Constructed Sorts

Non-standard data types may be specified algebraically [21, 47]. Beside such *abstract* types we also provide *constructed* types by means of predefined sort constructors. Here we propose the constructors *set*, *list*, *bag* to describe multi-valued domains, *tuple* to describe composite domains, and *union* to describe alternative domains. Of course, other constructors could be added as well. Sort constructors can be applied iteratively to construct domains of any complexity.

Definition 3 (sort expressions). Let a set of sorts S be given. Then the set $S\text{-}Expr(S)$ of sort expressions over S is defined as follows.

i. If $s \in S$, then $\boldsymbol{s} \in S\text{-}Expr(S)$.
ii. If $s \in S\text{-}Expr(S)$, then $\mathbf{set}(\boldsymbol{s}), \mathbf{list}(\boldsymbol{s}), \mathbf{bag}(\boldsymbol{s}) \in S\text{-}Expr(S)$.
iii. If $s_1, \ldots, s_n \in S\text{-}Expr(S)$, then $\mathbf{tuple}(\boldsymbol{s_1}, \ldots, \boldsymbol{s_n}) \in S\text{-}Expr(S)$.
iv. If $s_1, \ldots, s_n \in S\text{-}Expr(S)$, then $\mathbf{union}(\boldsymbol{s_1}, \ldots, \boldsymbol{s_n}) \in S\text{-}Expr(S)$.

Let us assume that there is a fixed interpretation I of S according to Definition 2. Then the interpretation of sort expressions is defined as follows.

i. see Definition 2.
ii. $I(\text{set}(s)) := \mathcal{F}(I(s)) \cup \{\perp\}$.
$I(\text{list}(s)) := (I(s))^* \cup \{\perp\}$.
$I(\text{bag}(s)) := \mathcal{B}(I(s)) \cup \{\perp\}$.
iii. $I(\text{tuple}(s_1, \ldots, s_n)) := (I(s_1) \times \ldots \times I(s_n)) \cup \{\perp\}$.
iv. $I(\text{union}(s_1, \ldots, s_n)) := \{1{:}\underline{s}_1 \mid \underline{s}_1 \in I(s_1)\} \cup \ldots \cup \{n{:}\underline{s}_n \mid \underline{s}_n \in I(s_n)\} \cup \{\perp\}$.

□

An element $\underline{s} \in I(s)$ with $s \in S\text{-}Expr(S)$ is called a *complex value*. As in the case of atomic sorts a special null value belongs to the interpretation of each sort expression.

In tuple or union expressions attributes may be added to underline the meaning of components. For example,

```
tuple(Nodes:set(int),
      Edges:set(tuple(Start:int, End:int)))
```

describes a structure for storing graphs. In union expressions attributes may serve as tags to distinguish several alternatives. For example,

```
union(PrivateAddress: tuple(Num:int,
                            Street:string,
                            City:string),
      BusinessAddress:tuple(Institution:string,
                            Office:string,
                            City:string))
```

describes a structure for storing addresses. Attributes in tuple expressions can be used as projection operators on tuple values. Attributes in union expressions can be used to test for or to select a certain alternative of a union value. In the following we treat attributes in tuple or union expressions as optional items.

As demonstrated by the two examples, sort constructors can be used to define *concrete* non-standard data sorts. Interestingly, sort constructors have been also served as the basis for a wide range of *value-based* data models.

A *schema* of a value-based data model can generally be expressed by a tuple expression $s = \mathrm{tuple}(r_1{:}s_1,\ldots,r_m{:}s_m)$. The *state* of a database belonging to this schema is a complex value of sort s. Many variations exist with respect to what is allowed for the sort expressions s_i $(i = 1, \ldots, m)$.

- The **relational model** [17] restricts s_i to describe *tables* of the kind $\mathrm{set}(\mathrm{tuple}(a_1{:}d_1,\ldots,a_n{:}d_n))$ in which d_j denote atomic sorts.
- The **nested form (NF) models** [41] allow for sort expressions s_i with alternating set and tuple constructors so that *nested tables* can be represented.
- The **complex object (CO) model** [1, 3] does not impose any restriction on the structure of s_i.

2.3 Generic Functions

Sort expressions are usually associated with a large number of *generic* (or overloaded) functions. For instance, there are operations and predicates that

- *generate* complex values from simpler ones. For instance, `{ }` is used as a constructor for set values and `( )` is used as a constructor for tuple values.
- *decompose* complex values. For instance, $\mathtt{SEL}_{\mathrm{list}(s)} : \mathrm{list}(s) \times \mathrm{nat} \longrightarrow s$ selects an element from a list, $\mathtt{PRJ}^i_{\mathrm{tuple}(s_1,\ldots,s_n)} : \mathrm{tuple}(s_1, \ldots, s_n) \longrightarrow s_i$ selects the ith component of a tuple, and $\mathtt{PRJ}^i_{\mathrm{union}(s_1,\ldots,s_n)} : \mathrm{union}(s_1, \ldots, s_n) \longrightarrow s_i$ selects the ith alternative of a union value.
- *convert* complex values. For instance, $\mathtt{BTS}_{\mathrm{bag}(s)} : \mathrm{bag}(s) \longrightarrow \mathrm{set}(s)$ converts a bag to a set by eliminating duplicates and $\mathtt{LTB}_{\mathrm{list}(s)} : \mathrm{list}(s) \longrightarrow \mathrm{bag}(s)$ converts a list to a bag.
- *exploit* the distinct properties of certain kinds of sort expressions. For instance, $\mathtt{CNT}_{\mathrm{set}(s)} : \mathrm{set}(s) \longrightarrow \mathrm{nat}$ counts the elements in a set and $\mathtt{IN}_{\mathrm{set}(s)} : \mathrm{set(s)} \times s$ denotes the membership predicate.

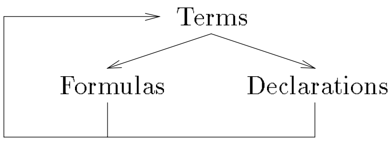
There are many other generic functions (for details see [25, 23, 27]). All operations induced by sort expressions are summarized in the set $\Omega(S\text{-}Expr(S))$. Analogously all predicates induced by sort expressions are combined in $\Pi(S\text{-}Expr(S))$.

From now on let $\Sigma_D = (S_D, \Omega_D, \Pi_D)$ denote a data signature according to Definition 1 so that $\Sigma = (S, \Omega, \Pi)$ denotes an *extended* signature given by $S = S\text{-}Expr(S_D)$, $\Omega = \Omega_D \cup \Omega(S\text{-}Expr(S_D))$, and $\Pi = \Pi_D \cup \Pi(S\text{-}Expr(S_D))$.

3 A Calculus of Complex Values

3.1 Formal Definition

In contrast to the well-known predicate calculus the calculus of complex values (in the following abbreviated as CCV) has a non-hierarchical structure given by



meaning that terms may be used to build formulas and declarations, and both formulas and declarations may appear in the construction of terms. In this respect CCV is derived from a calculus for querying extended Entity-Relationship schemas [25]. CCV is a revision of this query calculus. Firstly the calculus is lifted from a specific data model approach like the ER model to a general model of complex values, secondly we succeeded in drawing the syntactical parts of the calculus (esp. select terms and declarations) in simplified terms.

Terms may consist of variables. For the evaluation of such terms we need a notion of variable assignment.

Definition 4 (variables and variable assignments). Let a family $Var = \{Var_s\}_{s \in S}$ of S-indexed variables be given. The set of variable assignments A is defined by $A := \{\alpha \mid \alpha : Var_s \to I(s) \text{ for all } s \in S\}$. The special assignment $\epsilon : Var \to \{\perp\}$ is called the *empty assignment*. □

Definition 5 (terms). The syntax of terms is given by an S-indexed family $Term = \{Term_s\}_{s \in S}$ and a function free : $Term \to \mathcal{F}(Var)$ defined by the following rules.

i. If $v \in Var_s$, then $\boldsymbol{v} \in Term_s$ with $\text{free}(v) := \{v\}$.
ii. If $\omega : s_1 \times \ldots \times s_n \to s \in \Omega$ and $\tau_i \in Term_{s_i}$ $(i = 1 \ldots n)$, then $\boldsymbol{\omega(\tau_1, \ldots, \tau_n)} \in Term_s$ with $\text{free}(\omega(\tau_1, \ldots, \tau_n)) := \text{free}(\tau_1) \cup \ldots \cup \text{free}(\tau_n)$.
iii. If $\varphi \in Form$ and $\tau_1, \tau_2 \in Term_s$, then $\boldsymbol{?(\varphi, \tau_1, \tau_2)} \in Term_s$ with $\text{free}(?(\varphi, \tau_1, \tau_2)) := \text{free}(\varphi) \cup \text{free}(\tau_1) \cup \text{free}(\tau_2)$.
iv. If $\tau \in Term_s$, $\delta \in Decl$ and $\varphi \in Form$, then $\boldsymbol{\{\!\mid \tau \mid \delta,\ \varphi \mid\!\}} \in Term_{\text{bag}(s)}$ with $\text{free}(\{\!\mid \tau \mid \delta,\ \varphi \mid\!\}) := (\text{free}(\tau) \cup \text{free}(\delta) \cup \text{free}(\varphi)) \setminus \text{decl}(\delta)$.

For a fixed interpretation I and a variable assignment $\alpha \in A$ the evaluation of terms is defined as follows.

i. $(I, \alpha)[\![v]\!] = \alpha(v)$.
ii. $(I, \alpha)[\![\omega(\tau_1, \ldots, \tau_n)]\!] = I(\omega)((I, \alpha)[\![\tau_1]\!], \ldots, (I, \alpha)[\![\tau_n]\!])$.
iii. $(I, \alpha)[\![?(\varphi, \tau_1, \tau_2)]\!] = \begin{cases} (I, \alpha)[\![\tau_1]\!] & \text{if } (I, \alpha) \models \varphi \\ (I, \alpha)[\![\tau_2]\!] & \text{otherwise} \end{cases}$.

iv. $(I,\alpha)[\![\{\!\!\{ \tau \mid \delta,\ \varphi \}\!\!\}]\!] = \{\!\!\{ (I,\alpha')[\![\tau]\!] \mid \alpha' \in A$ with $\alpha'(v) = \alpha(v)$ for all $v \in \mathit{Var} \setminus \mathrm{decl}(\delta)$ and $(I,\alpha') \models \delta$ and $(I,\alpha') \models \varphi \}\!\!\}$. □

Variables (i), operation symbols (ii), and if-terms (iii) are standard. Terms (iv) are called *select terms*, because $\{\!\!\{ \tau \mid \delta,\ \varphi \}\!\!\}$ is merely a short-hand notation for *select* τ *from* δ *where* φ. As known from SQL select terms are the basic primitive for declarative queries. τ represents the target term fixing the format of the desired result, δ is a declaration binding one or more variables to finite domains (see below), and φ is a qualifying formula.

Example (terms in concrete syntax). `2`, `2+3*7` (infix notation of operation symbols is frequently used, operations have the usual bindings), `2+x`, `{1,4,9}` (the curly brackets serve as constructors for set values), `IF x<0 THEN -x ELSE x`, and `SELECT Name(x) FROM x IN PERSON WHERE Age(x)<18` are possible terms.

Terms are evaluated to complex values. With respect to the evaluation of operations and the evaluation of predicates in formulas, term evaluation depends on the interpretation I of the signature Σ. With respect to the evaluation of variables, term evaluation depends on the current variable assignment α. The evaluation of select terms is further discussed in Section 3.2 after having introduced formulas and declarations.

Definition 6 (formulas). The syntax of formulas is defined by a set *Form* and a function free : $\mathit{Form} \to \mathcal{F}(\mathit{Var})$ defined by the following rules.

i. If $\pi : s_1 \times \ldots \times s_n \in \Pi$ and $\tau_i \in \mathit{Term}_{s_i}$ $(i = 1 \ldots n)$, then $\boldsymbol{\pi(\tau_1, \ldots, \tau_n)}$ $\in \mathit{Form}$ with $\mathrm{free}(\pi(\tau_1,\ldots,\tau_n)) := \mathrm{free}(\tau_1) \cup \ldots \cup \mathrm{free}(\tau_n)$.
ii. If $\tau_1, \tau_2 \in \mathit{Term}_s$, then $\boldsymbol{\tau_1 = \tau_2} \in \mathit{Form}$ with $\mathrm{free}(\tau_1 = \tau_2) := \mathrm{free}(\tau_1) \cup \mathrm{free}(\tau_2)$.
iii. If $\varphi \in \mathit{Form}$, then $\boldsymbol{\neg(\varphi)} \in \mathit{Form}$ with $\mathrm{free}(\neg(\varphi)) := \mathrm{free}(\varphi)$.
iv. If $\varphi_1, \varphi_2 \in \mathit{Form}$, then $\boldsymbol{(\varphi_1 \vee \varphi_2)} \in \mathit{Form}$ with $\mathrm{free}((\varphi_1 \vee \varphi_2)) := \mathrm{free}(\varphi_1) \cup \mathrm{free}(\varphi_2)$.
v. If $\delta \in \mathit{Decl}$ and $\varphi \in \mathit{Form}$, then $\boldsymbol{\exists\delta(\varphi)} \in \mathit{Form}$ with $\mathrm{free}(\exists\delta(\varphi)) := (\mathrm{free}(\delta) \cup \mathrm{free}(\varphi)) \setminus \mathrm{decl}(\delta)$.

For a fixed interpretation I and a variable assignment $\alpha \in A$ the validity of formulas is defined as follows.

i. $(I,\alpha) \models \pi(\tau_1,\ldots,\tau_n)$ iff $((I,\alpha)[\![\tau_1]\!], \ldots, (I,\alpha)[\![\tau_n]\!]) \in I(\pi)$.
ii. $(I,\alpha) \models \tau_1 = \tau_2$ iff $(I,\alpha)[\![\tau_1]\!] = (I,\alpha)[\![\tau_2]\!]$.
iii. $(I,\alpha) \models \neg(\varphi)$ iff not $(I,\alpha) \models \varphi$.
iv. $(I,\alpha) \models (\varphi_1 \vee \varphi_2)$ iff $(I,\alpha) \models \varphi_1$ or $(I,\alpha) \models \varphi_2$.
v. $(I,\alpha) \models (\exists\delta(\varphi))$ iff there is a variable assignment $\alpha' \in A$ with $\alpha'(v) = \alpha(v)$ for all $v \in \mathit{Var} \setminus \mathrm{decl}(\delta)$ and $(I,\alpha') \models \varphi$ and $(I,\alpha') \models \delta$. □

There is nothing special about formulas. Note that the constant formulas `TRUE` and `FALSE`, the other logical connectives $\wedge$, $\Rightarrow$, $\Leftrightarrow$, and the quantifier $\forall$ might be defined by the given material.

With the aggregate functions CNT it is always possible to replace the formula $\exists\delta(\varphi)$ by CNT$\{\!\!\{ c \mid \delta, \varphi \}\!\!\} \geq 1$ (in which c represents an arbitrary constant term). In this respect quantifiers can be regarded as syntactic sugar [23].

Example (formulas in concrete syntax). `x<4` (infix notation of predicate symbols is frequently used), `x=y`, `NOT x=y`, `x=y AND x=z OR y=z` (logical connectives have the usual bindings), and `EXISTS p IN PERSON WHERE Name(p)="Smith"` are possible formulas.

Definition 7 (declarations). The syntax of declarations is given by a set *Decl* and the functions free, decl : $Decl \rightarrow \mathcal{F}(Var)$ defined by the following rules.

i. If $v \in Var_s$, $\tau \in Term_{\mathrm{set}(s)}$ and $v \notin \mathrm{free}(\tau)$, then $(\boldsymbol{v : \tau}) \in Decl$, $\mathrm{free}((v : \tau)) := \mathrm{free}(\tau)$ and $\mathrm{decl}((v : \tau)) := \{v\}$.
ii. If $v \in Var_s$, $\tau \in Term_{\mathrm{set}(s)}$, $\delta \in Decl$ and $v \notin \mathrm{free}(\tau) \cup \mathrm{free}(\delta) \cup \mathrm{decl}(\delta)$, then $(\boldsymbol{v : \tau;\ \delta}) \in Decl$, $\mathrm{free}((v : \tau;\ \delta)) := (\mathrm{free}(\tau) \setminus \mathrm{decl}(\delta)) \cup \mathrm{free}(\delta)$ and $\mathrm{decl}((v : \tau;\ \delta)) := \{v\} \cup \mathrm{decl}(\delta)$.

For a fixed interpretation I and a variable assignment $\alpha \in A$ the validity of declarations is defined as follows.

i. $(I, \alpha) \models (v : \tau)$ iff $\alpha(v) \in (I, \alpha)[\![\tau]\!]$.
ii. $(I, \alpha) \models (v : \tau;\ \delta)$ iff $\alpha(v) \in (I, \alpha)[\![\tau]\!]$ and $(I, \alpha) \models \delta$. □

Following the definition declarations are generally given as *declaration sequences* of the kind $(x_1{:}\tau_1;\ (x_2{:}\tau_2;\ \ldots;\ (x_n{:}\tau_n)))$. Declarations are used in select terms or in quantified formulas to bind variables to finite sets of values.

Example (declarations in concrete syntax). `x IN {1,2,3}`, `p IN PERSON` (in which `PERSON` denotes a set-valued term), `p IN PERSON; c IN CAR`, and `h IN Hobbies(p); p IN PERSON` (in which `Hobbies(p)` denotes a set-valued attribute of persons) are possible declarations.

Finally we define queries as closed terms, i.e., terms without free variables.

Definition 8 (queries). Every term $\tau \in Term$ with $\mathrm{free}(\tau) = \emptyset$ defines a query. The evaluation of queries is determined by $(I, \epsilon)[\![\tau]\!]$. □

In the following brackets are dropped as far as unambuigity is preserved. Infix notations are used where appropriate. $\{\!\!\{ \tau \mid \delta \}\!\!\}$ is used as a short-hand for $\{\!\!\{ \tau \mid \delta,\ TRUE \}\!\!\}$.

3.2 Evaluation of Select Terms

Terms are evaluated like applicative programs — may be except for select terms. A select term of the general kind

$$\{\!\!\{\ \tau \mid x_1{:}\tau_1;\ x_2{:}\tau_2;\ \ldots;\ x_n{:}\tau_n,\ \varphi\ \}\!\!\}$$

is evaluated by examining all possible assignments of values to the variables $x_1, \ldots, x_n$. Variable assignments have to fulfill the declarations and the qualifying formula φ. Having found a possible assignment this assignment contributes to the query result by evaluating the term τ. Assuming all terms τ_i to be evaluated to finite sets there can only exist a finite number of possible assignments to variables $x_1, \ldots, x_n$. Hence evaluation of select terms always results in finite sets. The query calculus is *safe* [45] (which can be formally proved by induction over term construction [25]). Importantly there is no need to define safe queries in a semantic way as in the case of relational domain calculus.

There is a simple scheme of describing the semantics of a declarative select term by an imperative program. More precisely a select term of the above kind can be evaluated by the following procedure:

result := $\{\!\{\ \}\!\}$;
for each $x_n \in [\![\, \tau_n \,]\!]$ **do**
  ...
    for each $x_1 \in [\![\, \tau_1 \,]\!]$ **do**
      if $\models \varphi$ **then** result := result $\cup$ $[\![\, \tau \,]\!]$ **fi**;
    od;
  ...
od;

The order of the for-each statements reflects the scope of variables in declaration sequences extending from the right to the left.

Next we try to explain the semantics of select terms by giving some examples.

Examples.

1. The first example demonstrates the basic *filter* function of select terms.

   `-{ x | x:{-2,-1,0,1,2}, x>0 }-`

   The term is evaluated to $\{\!\{\, 1, 2 \,\}\!\}$.
2. In general select terms are evaluated to true bags. For instance, the term

   `-{ x^2 | x:{-2,-1,0,1,2} }-`

   results in $\{\!\{\, 0, 1, 1, 4, 4 \,\}\!\}$. The aggregate function `BTS` can be used to convert bags into sets by eliminating duplicates.
3. Declarations may be given as declaration sequences. This can be used to describe the *cross product*. For example

   `-{ (x,y) | x:{1,2}; y:{1,2,3} }-`

   is evaluated to $\{\!\{\, (1,1),\ (1,2),\ (1,3),\ (2,1),\ (2,2),\ (2,3) \,\}\!\}$. Together with a qualifying formula φ this allows the formulation of *joins*.
4. In a declaration sequence like $\mathrm{x}_1{:}\tau_1;\ldots;x_n{:}\tau_n$ the variable x_i is allowed to be free in $\tau_1, \ldots, \tau_{i-1}$. This can be used to express the *union* of sets being demonstrated by

```
⦃ x | x:y; y:{{1,2}, {2,3}} ⦄
```

which is evaluated to ⦃ *1, 2, 2, 3* ⦄. By applying the function `BTS` the result can be converted into a proper set.

5. With nested select terms *groupings* can be expressed. This is shown by the next example which groups the flat value {(*1, 2*), (*1, 3*), (*2, 3*)} by the first component resulting in {(*1*, {*2, 3*}), (*2*, {*3*})} (x.i is used as an abbreviation for $\texttt{PRJ}^i$(x)).

```
BTS⦃ (x.1,BTS⦃ y.2 | y:{(1,2),(1,3),(2,3)}, y.1=x.1 ⦄) |
      x:{(1,2),(1,3),(2,3)} ⦄
```

6. The result of the last select term can be *unnested* again by the select term

```
BTS⦃ (x.1,y) | y:x.2; x:{(1,{2,3}), (2,{3})}⦄
```

yielding the original value.

3.3 Properties

Comparison with Query Algebras. The calculus allows to express all standard operations on complex values. This is shown in Table 1. The first group of operations subsumes the classical operations of relational algebra, showing that CCV is at least *relationally complete*. In the second group ν denotes the *nest* and μ the *unnest* operator of [44]. Specific operations for CO models can be expressed as well. For example, *set construction* ν' and *set collapse* μ' can be derived from the corresponding ν and μ representations by dropping the first tuple component in both cases.

Table 1. Standard operations on complex values[1]

$\cup$	$\text{set}(s) \times \text{set}(s) \rightarrow \text{set}(s)$	⦃ x | x:y; y:{τ_1, τ_2} ⦄
$-$	$\text{set}(s) \times \text{set}(s) \rightarrow \text{set}(s)$	⦃ x | x:τ_1, ¬IN(τ_2,x) ⦄
$\times$	$\text{set}(s_1) \times \text{set}(s_2) \rightarrow \text{set}(\text{tuple}(s_1, s_2))$	⦃ (x,y) | x:τ_1; y:τ_2 ⦄
π_i	$\text{set}(\text{tuple}(s_1, \ldots, s_n)) \rightarrow \text{set}(s_i)$	⦃ x.i | x:τ ⦄
σ_φ	$\text{set}(s) \rightarrow \text{set}(s)$	⦃ x | x:τ, φ ⦄
ν	$\text{set}(\text{tuple}(s_1, s_2)) \rightarrow \text{set}(\text{tuple}(s_1, \text{set}(s_2)))$	⦃ (x.1, ⦃ y.2 | y:τ, y.1 = x.1⦄) | x:τ ⦄
μ	$\text{set}(\text{tuple}(s_1, \text{set}(s_2))) \rightarrow \text{set}(\text{tuple}(s_1, s_2))$	⦃ (x.1,y) | y:x.2; x:τ ⦄
ν'	$\text{set}(s) \rightarrow \text{set}(\text{set}(s))$	⦃ ⦃ y | y:τ, y = x ⦄ | x:τ ⦄
μ'	$\text{set}(\text{set}(s)) \rightarrow \text{set}(s)$	⦃ y | y:x; x:τ ⦄

It should be noted that all the operators found in Table 1 could also find their entrance into the query calculus as generic functions on complex values. In some cases this may assist the formulation of complex queries.

[1] In every calculus expression τ_i stands for the ith argument of the corresponding operation. The dot notation is used for projection. To allow comparison, here all select terms are assumed to evaluate to proper sets by implicit duplicate elimination.

Comparison with other Query Calculi. CCV stands in the tradition of query calculi proposed for models of complex values [37, 10, 43, 2] from which the calculus of Abiteboul and Beeri (denoted CALC), which was studied in more detail in [1], can be considered as an archetype. Comparing CCV with CALC we find the following distinguishing features:

1. The consideration of *non-logical components* (data functions, aggregate functions) as important ingredients of concrete query languages.
2. The treatment of *duplicates* in the evaluation of select terms.
3. The *non-hierarchical structure* of the query calculus guided by the idea of fully orthogonal SQL.

We will briefly discuss the three points in more detail.

ad 1. The incorporation of externally defined functions in the query calculus clearly does not contribute to its inherent computational power. This was the main reason why such functions were not considered in [1]. On the other hand the pure calculus, even augmented with a fixpoint operator for expressing recursive queries (see below on expressiveness), still fails in solving such a trivial problem like testing a given finite set for even cardinality (*even problem* [6]). It has been argumented in [7] that the role of a query language should be primarily the selection and combination of data from a database, rather than arithmetic or other general computation on this data. So it should not be considered harmful when a query language does not show full computational power. Lack of expressive power can always be compensated by a careful combination of a query language with a general purpose programming language. Of course, then the main problem still consists in overcoming the well-known *impedance mismatch* between the set-oriented type system of a query language and the record-oriented style of conventional programming languages.

ad 2. In CALC a query is simply given by a formula φ with a single free variable v, for instance $\exists \mathrm{x}\ \mathrm{x} \in \{-2,-1,0,1,2\} \wedge \mathrm{x}^2 = v$ (compare with CCV query 2 above). The answer to such a query is the set of all possible assignments of values to v making the formula φ true (here $\{\mathit{0}, \mathit{1}, \mathit{4}\}$). It is clear that in such a way one cannot obtain duplicates in query results.

ad 3. Since queries in CALC are given as usual formulas it is nearly impossible to structure complex queries. The decreased readability of CALC queries in comparison with SQL-like queries has been criticized in [12]. It follows from the fact that the only way to deal with subqueries in CALC consists in using *set variables*. For example, the following CALC query with v as single free variable and a as a set variable corresponds to the grouping example (query 5) of the CCV queries given above (let M stand for `{(1,2),(1,3),(2,3)}`, duplicate handling neglected).

$$\exists \mathrm{x}\ \mathrm{x} \in \mathrm{M} \wedge \exists \mathrm{a}\ \big((\forall \mathrm{t}\ \mathrm{t} \in \mathrm{a} \Leftrightarrow \exists \mathrm{y}\ \mathrm{y} \in \mathrm{M} \wedge \mathrm{y}.1 = \mathrm{x}.1 \wedge \mathrm{t} = \mathrm{y}.2) \wedge v = (\mathrm{x}.1, \mathrm{a})\big)$$

Since in CCV every select term is nothing else than a special kind of term, select terms may appear in any place as subterms of other select terms achieving full orthogonality for composing queries from subqueries.

Expressiveness. There exist two variants of CALC:

1. CALC^- where like in CCV all variables must always be bound to constant, stored or computed sets, and
2. unrestricted CALC where variables may directly range over data domains.

It has been shown in [27] that, when ignoring data functions and duplicate handling, every CCV query can be transformed into a corresponding CALC^- query by making use of set variables for representing results of subqueries.

It is important to note that CALC is more expressive than CALC^-. Like the relational calculus and the calculus presented in this paper CALC^- lacks the power of expressing the computation of the transitive closure of a relation. It is shown in [1] that in a query algebra a special power set operator is needed to express transitive closure. Notably, power set of M can be expressed in CALC by $\forall \mathrm{x}\ \mathrm{x} \in v \Rightarrow \mathrm{x} \in \mathrm{M}$ where v represents the only free variable. Unfortunately, the so-called *domain-dependent* calculus CALC is not safe (take for example the query $v \neq 42$). Moreover, using power set to compute transitive closure is not very efficient at all.

Transitive closure can be easily described by *recursion* in a rule-based framework. Hence another idea is to include a *fixpoint operator* μ into the calculus (see [36] for a concrete language proposal). One can show that first-order calculus with inflationary fixpoint operator is equivalent to algebra with inflationary *while* and Datalog with negation (see [5, 6] for details).

Here we follow a more pragmatic approach based on an extensible signature Σ. Whenever a special operator like power set $\mathtt{POW}_{\mathrm{set}(s)} : \mathrm{set}(s) \rightarrow \mathrm{set}(\mathrm{set}(s))$ or transitive closure $\mathtt{TC}_{\mathrm{set}(\mathrm{tuple}(s,s))} : \mathrm{set}(\mathrm{tuple}(s,s)) \rightarrow \mathrm{set}(\mathrm{tuple}(s,s))$ is needed this can always be defined *externally* by a corresponding generic function.

4 Applications

The calculus of Section 3 was developed independently of any concrete database model. Queries were formulated over data constants instead. Clearly in real applications we want to apply the calculus to stored values. Therefore in this section we first discuss the application of the calculus to existing database models. Secondly we give some insight into an approach to object-oriented specification where the query calculus is used in the formulation of axioms expressing object properties.

4.1 Queries in Value- and Object-Based Data Models

A database schema generally includes a number of *containers* for storing objects that share common structure and common behavior. In value-based data models these containers are given by *relations* (or *relational schemas*); in object-based data models they are given by *classes*. The fundamental difference between value-based and object-based data models lies in their particular approach to the representation of real-world entities [31]. While in value-based models a real-world entity is represented directly as a tuple of its attribute values (*internal structure*), object-based data models spend a special object identifier (surrogate) for this purpose and attributes are functions on object identifiers (*attribute structure*). The latter approach enables updates on objects.

Referring to stored values in queries is based on references to containers. Hence a simple way of extending the query calculus to database states consists in including containers as set-valued terms into the logic. In addition, in object-based data models attributes must be admitted as functions on object surrogates. This is summarized in Table 2.

Table 2. Including containers into the query calculus

value-based	relation $r(a_1{:}s_1,\ldots,a_n{:}s_n)$	$r \in \mathit{Term}_{\mathrm{set}(\mathrm{tuple}(s_1,\ldots,s_n))}$
object-based	class c with $\mathrm{osort}(c) = o$	$c \in \mathit{Term}_{\mathrm{set}(o)}$
	attribute $a : o \to s$	$a \in \Omega$

The first line of the table also holds for the extensions of the relational model mentioned in Section 2.2. In the object-based approach we strictly distinguish between a class c and a sort of object identifiers $\mathrm{osort}(c)$ belonging to a class c although in many data model approaches same names are used to denote both classes and corresponding object identifier sorts.

Dependent on the current state of a database, containers are evaluated to the current set of tuple values (or object identifiers respectively). Also in a state-dependent way attribute functions return the current attribute values.

Examples.

1. Let a relational schema `PERSON(Name:string, Age:nat)` be given. Then the following query returns the names of all stored persons:

   $\{\!\!\{$ `Name(p) | p:PERSON` $\}\!\!\}$

   or in concrete syntax:

   ```
   SELECT Name(p) FROM p IN PERSON
   ```

   In this query p ist a variable of sort tuple(Name:string, Age:nat) so that Name acts as an projection operator.
2. Let us replace the relational schema by a class `PERSON` with attributes `Name:string` and `Age:nat`. A query returning the names of all stored persons is again given by $\{\!\!\{$ `Name(p) | p:PERSON` $\}\!\!\}$. But unlike in the former

query here p is a variable of object identifier sort person = osort(PERSON) and Name is used as an attribute function. Note that both select terms are of sort bag(string) reflecting the fact that a given name may appear more than once in a query result. Note also that in the object-based database it is even possible that two persons with the same name and the same age exist while this is excluded in the corresponding relational database.
Next we assume all queries being formulated in context of the object-based database.

3. The following query gives an example of a select term with a nested select in the select-clause (grouping).

```
BTS{ (p1, BTS{ p2 | p2:PERSON, Age(p2)>Age(p1) }) | p1:PERSON }
```

   or in concrete syntax:

```
BTS(SELECT (p1, BTS(SELECT p2
                    FROM   p2 IN PERSON
                    WHERE  Age(p2)>Age(p1)))
    FROM   p1 IN PERSON)
```

   The query results in a value of sort set(tuple(person, set(person))). It contains all persons together with the older ones.

4. The next query gives an example of a nested select in the where-clause of a select term.

```
BTS{ x | x:PERSON, Age(x)=MAX{ Age(y) | y:PERSON } }
```

   or in concrete syntax:

```
BTS(SELECT x
    FROM   x IN PERSON
    WHERE  Age(x)=MAX(SELECT Age(y)
                      FROM   y IN PERSON))
```

   The query returns the set of the oldest persons. Note that in concrete syntax we did not write `SELECT MAX(Age(y))` as it would have been done in standard SQL showing some peculiar inconsistency of this standard [20].
   The same query could also be formulated without making use of an aggregate function by:

```
BTS{ x | x:PERSON, ∀ y:PERSON Age(y)<=Age(x) }
```

5. Let <Q4> denote the query given in item 4. Then the following query returns the names of the oldest persons.

```
BTS{ Name(p) | p:<Q4> }
```

   This also gives an example for a query with nested select in the from-clause of a select term.

6. Let `Hobbies:SET(string)` be a further attribute of `PERSON`. Then the following query which makes use of a declaration sequence returns the set of hobbies of persons with age less than 18.

   ```
   BTS{| h | h:Hobbies(p); p:PERSON, Age(p)<18 |}
   ```

   or in concrete syntax:

   ```
   BTS(SELECT h
       FROM   h IN Hobbies(p); p IN PERSON
       WHERE  Age(p)<18)
   ```

4.2 Axioms in Object-Oriented Specifications

The calculus of complex values was motivated by a concrete language proposal for describing static and dynamic properties of objects called TROLL *light* [18, 24, 28, 27]. For TROLL *light* a sublanguage was needed for the axiomatic part of object descriptions called *templates*. This includes static integrity constraints, rules to describe the effect of events on attributes, derivation rules for attributes, rules for the synchronization of events in different objects, event preconditions, etc.

Example.

```
TEMPLATE Node
  DATA TYPES   String, Nat;
  SUBOBJECTS   Children(Name:string):node;
  ATTRIBUTES   Info:nat;
               DERIVED Total:nat;
  EVENTS       BIRTH create(nat);
                     updateInfo(nat);
                     createChild(Name:string, Info:nat);
               DEATH destroy;
  CONSTRAINTS  CNT(Children) < 5;
  VALUATION    [create(I)]     Info = I;
               [updateInfo(I)] Info = I;
  DERIVATION   Total = Info + SUM(SELECT Total(x)
                                  FROM x IN Children);
  INTERACTION  createChild(N,I) >> Children(N).create(I);
  BEHAVIOR     PROCESS Node =
                 ( create -> Nodelife );
               PROCESS Nodelife =
                 ( updateInfo -> Nodelife  |
                   createChild -> Nodelife |
                   {CNT(Children)=0} destroy );
END TEMPLATE;
```

The template describes nodes (of a tree). Every node is assigned a natural number as node information. Each node may have up to four subnodes being identified by a string. For a given node n the derived attribute Total returns the sum of all node information found in the subtree rooted at n (*bill of material* problem). Besides the birth and death of a node other possible events in a node's lifecycle concern updates of the node information and adding of subnodes. A node may only die when currently there are not any subnodes.

Templates may describe object communities as trees of arbitrary width and depth. In this special case the object community is a homogeneous one with all objects being of the same sort. In general object communities will be composed of rather different kinds of objects.

Let us now pay closer attention to how the query calculus is used in the construction of templates. Here we focus on the description of *derived attributes* which may be considered as predefined queries and the formulation of (static) *integrity constraints* which may be regarded as invariants.

It is a distinctive feature of TROLL *light* object descriptions that they do not presume a certain global schema view as this is usually done in most value- or object-based data models. This follows from the fact that with TROLL *light* descriptions of complex object systems can be iteratively composed from descriptions of subsystems using the subobject concept of templates [28]. This has a certain impact on the manner how queries are formulated against object communities. To summarize the difference: Queries are formulated from the perspective of local objects rather than starting from a fixed schema level.

Table 3, being a continuation of Table 2, shows the technical details to deal with queries in context of TROLL *light*.

Table 3. Support for queries formulated in the context of objects

TROLL *light*	template t with $\mathrm{osort}(t) = o$	$\mathrm{self} \in \mathit{Term}(t)_o$
	attribute $a : o \rightarrow s$	$a \in \Omega$
	subobject slot $u : o \rightarrow s$	$u \in \Omega$

Instead of referring to (global) containers, every query to be formulated in context of a certain object $\underline{o}$ starts with a self-reference to this object. Depending on the template t of which this object is an instance, the self-term to be evaluated is of object identifier sort $\mathrm{osort}(t)^2$ and, as you might already guess, in context of $\underline{o}$ self is evaluated to $\underline{o}$.

In concrete syntax the self-term is often omitted. To show the difference we give the abstract conterpart of the derivation rule.

$\texttt{Info(self)+SUM}\{\!\!\{ \texttt{ Total(PRJ}^2\texttt{(x)) | x:Children(self) } \}\!\!\}$

[2] In TROLL *light* there is the following convention: Names of templates start with an upper case letter while same names but with a lower case letter are used for the corresponding object identifier sorts.

Having the reference to an object one may firstly observe its attribute or subobject slots. Both are considered as functions on object identifier sorts. Attributes are evaluated to the current attribute values. Subobject symbols can be parametrized but they need not. A non-parametrized subobject symbol is evaluated to the object identifier of the corresponding subobject if this does currently exist. Otherwise it is evaluated to the error value $\perp$. Parametrized subobject symbols are treated as map-valued functions. For instance, in the above example Children is treated as a function $\mathrm{Children} : \mathrm{node} \rightarrow \mathrm{map}(\mathrm{string}, \mathrm{node})$. Such a function yields the set of object identifiers of all existing subobjects together with their logical names in the superobject.

Now to explain the above query in full detail: Children(self) is a term of map(string,node) being evaluated to the set of current subnodes together with their names. In turn map(string,node) can be regarded as a subsort of set(tuple(string,node)) with a functional dependence of the second component onto the first. Hence x is a variable of sort tuple(string,node) so that $\mathrm{Total}(\mathrm{PRJ}^2(\mathrm{x}))$ returns the Total value of a subnode (in concrete syntax Total(x) has to be considered as an abbreviation of the full path expression). All Totals of subnodes are summed up and the local Info value is added to give the desired result.

With this information in mind the constraint expressed in the node template should be self-explaining. It is important to note that in the cited form the query calculus can only be used to express static integrity conditions (invariants). Apart from other means for behavior specification transitional integrity conditions could be expressed by adding special *old* and *new* predicates referring to the state before or after the current state. For arbitrary temporal constraints temporal logic may be employed.

5 Conclusions

In this paper we gave a formal definition of a query calculus in the context of complex values. We showed that this calculus can be easily applied to both value-based and object-based database models. Moreover it provides a general facility for the manipulation of complex values. For instance, it can be used for stating axioms in object-oriented specifications.

Regarding general issues of object-oriented query languages [13] we claim that our query calculus fulfills basic requirements. For instance, *navigation* is supported by having surrogate-valued attributes as operations, and *methods* are integrated by user-defined operations and by derived attributes. The calculus is *extensible* by changes to the basic signature. We stress the fact that in contrast to many other query proposals our query calculus is completely *orthogonal*. This can smooth out some need to explicitly store interim query results.

Object-oriented query languages have been classified along the lines of *value-generating*, *object-generating* and *object-preserving*. With respect to this classification the presented calculus may be called value-generating. (However, it is a characteristic feature of values in contrast to objects that their existence is time-independent. Only objects can be created or deleted [40]. So a better denomination would be *value-resulting*.) When restricting select queries to select terms of sort set(o) where o represents an object identifier sort the corresponding query calculus could also be called object-preserving. Queries based on the invention of new object identifiers were not considered in this paper (cf. [4]).

A prototype implementation of the calculus using the facilities of the OODBMS ObjectStore [38] to store complex values has been finished. The term & formula evaluation unit is part of a validation tool for object-oriented specifications [46, 29, 27]. The aim of this tool is to support the animation of object descriptions according to [18]. At the time being term evaluation is based on the (naive) evaluation scheme of Section 3.2 which does not take into account query optimization [33]. Since query evaluation is close to evaluation of applicative programs, optimization strategies developed in this field may be adopted for our approach.

References

1. S. Abiteboul and C. Beeri, *On the Power of Languages for the Manipulation of Complex Objects*, Research report 846, INRIA France, 1988.
2. S. Abiteboul, C. Beeri, M. Gyssens, and D. Van Gucht, *An Introduction to the Completeness of Languages for Complex Objects and Nested Relations*, In Abiteboul et al. [3], pp. 117–138.
3. S. Abiteboul, P.C. Fischer, and H.J. Schek (eds.), *Nested Relations and Complex Objects in Databases*, Springer, Berlin, LNCS 361, 1989.
4. S. Abiteboul and P. Kanellakis, *Object Identity as a Query Language Primitive*, Proc. ACM Int. Conf. on Management of Data (SIGMOD) (J. Clifford, B. Lindsay, and D. Maier, eds.), ACM SIGMOD Record 18:2, 1989, pp. 159–173.
5. S. Abiteboul and V. Vianu, *Datalog Extensions for Database Updates and Queries*, Research report 715, INRIA France, 1988.
6. ———, *Expressive Power of Query Languages*, Research report 1587, INRIA France, 1992.
7. A.V. Aho and J.D. Ullman, *Universality of Data Retrieval Languages*, Proc. 6th ACM Symp. Principles of Programming Languages (POPL), 1979, pp. 110–120.
8. F. Bancilhon, S. Cluet, and C. Delobel, *A Query Language for the O_2 Object-Oriented Database System*, Proc. 2nd Int. Workshop on Database Programming Languages (R. Hull, R. Morrison, and D. Stemple, eds.), Morgan-Kaufmann, San Mateo (CA), 1989, pp. 122–138.
9. F. Bancilhon, C. Delobel, and P. Kanellakis (eds.), *Building an Object-Oriented Database System - The Story of O_2*, Morgan-Kaufmann, San Mateo (CA), 1992.
10. F. Bancilhon and S. Khoshafian, *A Calculus of Complex Objects*, Proc. 5th ACM Symp. Principles of Database Systems (PODS), 1986, pp. 53–60.
11. D. Beech, *A Foundation for Evolution from Relational to Object Databases*, Advances in Database Technology, Proc. Int. Conf. on Extending Database Technology (EDBT) (J.W. Schmidt, S. Ceri, and M. Missikoff, eds.), Springer, Berlin, LNCS 303, 1988, pp. 256–270.
12. C. Beeri, *A Formal Approach to Object-Oriented Databases*, Data & Knowledge Engineering **5** (1990), no. 4, 353–382.

13. E. Bertino, M. Negri, G. Pelagatti, and L. Sbattella, *Object-Oriented Query Languages: The Notion and the Issues*, IEEE Trans. on Knowledge and Data Engineering **4** (1992), no. 3, 223–237.
14. J. Van den Bussche and A. Heuer, *Using SQL with Object-Oriented Databases*, Information Systems **18** (1993), no. 7, 461–487.
15. R. Cattell, *The Object Database Standard: ODMG-93*, Morgan-Kaufmann, San Mateo (CA), 1994.
16. S. Cluet, C. Delobel, C. Lécluse, and P. Richard, *RELOOP, an Algebra Based Query Language for an Object-Oriented Database System*, Data & Knowledge Engineering **5** (1990), no. 4, 333–352.
17. E.F. Codd, *A Relational Model of Data for Large Shared Data Banks*, Communications of the ACM **13** (1970), no. 6, 377–387.
18. S. Conrad, M. Gogolla, and R. Herzig, *TROLL light: A Core Language for Specifying Objects*, Informatik-Bericht 92–02, Technische Universität Braunschweig, 1992.
19. S. Dar, N.H. Gehani, and H.V. Jagadish, *CQL++: A SQL for the Ode Object-Oriented DBMS*, Advances in Database Technology, Proc. Int. Conf. on Extending Database Technology (EDBT) (A. Pirotte, C. Delobel, and G. Gottlob, eds.), Springer, Berlin, LNCS 580, 1992, pp. 201–216.
20. C. Date, *A Critique of the SQL Database Language*, ACM SIGMOD Record **14** (1984), no. 3, 8–54.
21. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, Springer, Berlin, 1985.
22. G. Gardarin and P. Valduriez, *ESQL2: An Object-Oriented SQL with F-Logic Semantics*, Proc. 8th Int. Conf. on Data Engineering (ICDE), IEEE Computer Society Press, 1992, pp. 320–327.
23. M. Gogolla, *An Extended Entity-Relationship Model — Fundamentals and Pragmatics*, Springer, Berlin, LNCS 767, 1994.
24. M. Gogolla, S. Conrad, and R. Herzig, *Sketching Concepts and Computational Model of TROLL light*, Proc. 3rd Int. Conf. Design and Implementation of Symbolic Computation Systems (DISCO) (A. Miola, ed.), Springer, Berlin, LNCS 722, 1993, pp. 17–32.
25. M. Gogolla and U. Hohenstein, *Towards a Semantic View of an Extended Entity-Relationship Model*, ACM Trans. on Database Systems **16** (1991), no. 3, 369–416.
26. C. Harris and J. Duhl, *Object SQL*, Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD (R. Gupta and E. Horowitz, eds.), Prentice-Hall, 1991, pp. 199–215.
27. R. Herzig, *Zur Spezifikation von Objektgesellschaften mit TROLL light*, Ph.D. thesis, Technische Universität Braunschweig, 1994.
28. R. Herzig, S. Conrad, and M. Gogolla, *Compositional Description of Object Communities with TROLL light*, Proc. Basque Int. Workshop on Information Technology (BIWIT): Information Systems Design and Hypermedia (C. Chrisment, ed.), Cépaduès-Éditions, Toulouse, 1994, pp. 183–194.
29. R. Herzig and M. Gogolla, *An Animator for the Object Specification Language TROLL light*, Proc. Colloquium on Object Orientation in Databases and Software Engineering (V.S. Alagar and R. Missaoui, eds.), Université du Quebéc à Montréal, 1994, pp. 4–17.
30. U. Hohenstein and G. Engels, *SQL/EER — Syntax and Semantics of an Entity-Relationship-Based Query Language*, Information Systems **17** (1992), no. 3, 209–242.

31. R. Hull, *Four Views of Complex Objects: A Sophisticate's Introduction*, In Abiteboul et al. [3], pp. 87–116.
32. R. Hull and R. King, *Semantic Database Modelling: Survey, Applications, and Research Issues*, ACM Computing Surveys **19** (1987), no. 3, 201–260.
33. M. Jarke and J. Koch, *Query Optimization in Database Systems*, ACM Computing Surveys **16** (1984), no. 2, 111–152.
34. M. Kifer, W. Kim, and Y. Sagiv, *Querying Object-Oriented Databases*, Proc. ACM Int. Conf. on Management of Data (SIGMOD) (M. Stonebreaker, ed.), ACM SIGMOD Record 21:2, 1992.
35. W. Kim, *Observations on the ODMG-93 Proposal for an Object-Oriented Database Language*, ACM SIGMOD Record **23** (1994), no. 1, 4–9.
36. K. Koymen and Q. Cai, *SQL*: A Recursive SQL*, Information Systems **18** (1993), no. 2, 121–128.
37. G.M. Kuper and M.Y. Vardi, *A New Approach to Database Logic*, Proc. 3th ACM Symp. Principles of Database Systems (PODS), 1984, pp. 86–96.
38. C. Lamb, G. Landis, J. Orenstein, and D. Weinreib, *The ObjectStore Database System*, Communications of the ACM **34** (1991), no. 10, 50–63.
39. C. Lécluse and P. Richard, *Modeling Complex Structures in Object-Oriented Databases*, Proc. 8th ACM Symp. Principles of Database Systems (PODS), 1989, pp. 360–368.
40. B.J. MacLennan, *Values and Objects in Programming Languages*, ACM SIGPLAN Notices **17** (1982), no. 12, 70–79.
41. A. Makinouchi, *A Consideration on Normal Form of Not Necessarily Normalized Relation in the Relational Data Model*, Proc. 3rd Int. Conf. on Very Large Data Bases (VLDB), 1977, pp. 447–453.
42. M. Negri, G. Pelagatti, and L. Sbattella, *Formal Semantics of SQL Queries*, ACM Trans. on Database Systems **16** (1991), no. 3, 513–534.
43. M.A. Roth, H.F. Korth, and A. Silberschatz, *Extended Algebra and Calculus for Nested Relational Databases*, ACM Trans. on Database Systems **13** (1988), no. 4, 389–417.
44. H.J. Schek and M.H. Scholl, *The Relational Model with Relation-Valued Attributes*, Information Systems **11** (1986), 137–147.
45. J.D. Ullman, *Principles of Database and Knowledge Base Systems, Vol. I*, Computer Science Press, Rockville (MD), 1988.
46. N. Vlachantonis, R. Herzig, M. Gogolla, G. Denker, S. Conrad, and H.-D. Ehrich, *Towards Reliable Information Systems: The KORSO Approach*, Advanced Information Systems Engineering, Proc. 5th CAiSE'93 (C. Rolland, F. Bodart, and C. Cauvet, eds.), Springer, Berlin, LNCS 685, 1993, pp. 463–482.
47. M. Wirsing, *Algebraic Specification*, Handbook of Theoretical Computer Science, Vol. B (J. Van Leeuwen, ed.), North-Holland, Amsterdam, 1990, pp. 677–788.

This article was processed using the LaTeX macro package with LLNCS style