Incomplete Information in Object-Oriented Databases

Roberto Zicari
GIP Altaïr
and
Politecnico di Milano

January 8, 1990

Abstract

We present a way to handle incomplete information both at schema and object instance level in an object-oriented database. Incremental schema design becomes possible with the introduction of *generic* classes. Incomplete data in an object instance is handled with the introduction of explicit *null values* in a similar way as in the relational and nested relations data models.

1 Introduction

During the past years, the object-oriented approach has been extensively used to design programming languages and database systems.

Less attention has been paid to the problem of handling incomplete information in the object-oriented paradigm. In this paper, we address the issue of defining and managing incomplete information in the framework of an object-oriented database.

We have chosen the O_2 object-oriented database system as a representative example of an object-oriented database model as it includes most of the traditonal feautures [ACM TOOIS87, AtBu87, Ca88, OOPSLA87-88,DaTo88]. However, most of the considerations here presented for O_2 can be applied to other object-oriented database management systems as well.

We start with some basic definitions of the O_2 data model.

1.1 Preliminary O_2 concepts

O_2 is an object-oriented database system and programming environment developed at Altaïr. The functionalities of the system include those typical of a DBMS, those of a programming language and those of a programming environment. The O_2 systems is a complete development environment for data intensive applications.

O_2 is object-oriented [LeRi89a]: Informaton is organized as *objects* which have an identity and encapsulate data and behavior. Manipulation of objects is done through *methods*, which are procedure attached to objects. Typically in object-oriented systems, the value encapsulated in an object is a tuple or a set of objects [Maier83, Ba87a,Ba87b,LeRi89a].

O_2 provides two structuring data: Types and classes. Types are recursively contructed using atomic types such as integers, float, strings, class names and set, list, and tuple constructors. Types are attached to classes. A class describes the structure and the behav-

ior of a set of objects. The structural part of a class is the type associated to it, and the behavioral part is the set of methods. The distinction between the class concept and the type associated to a class is peculiar to the O_2 system. Other systems do not make this distinction. The advantage in having types is that O_2 offers a compile-type checker in an attempt to statically detect as many illegal manipulation as possible of objects and values.

In O_2, the schema is a set of classes related by inheritance links and/or composition links [LeRi89a,LeRi89b]. Classes are created using schema definition commands. An example of a class definition is the following:

```
add class City with extension
    type tuple (name:string,
                country:string,
                population:integer)
```

To each class there corresponds instances, that is objects with a unique internal identifier and a value which is an instance of the type associated with the class. An example of an instance of the class City is the following:

```
Milano = new(City)
*Milano = tuple (name: "Milano",
                 country: "Italy",
          population: 2.000.000)
```

Objects are encapsulated, their value are not directly accessible and they are manipulated by methods. Method definition is done in two steps: First the user declares the method by giving its signature, that is, its name, the type or class of the arguments and the type or class of the result (if any). Then the code of the method is given. An example of a method definition for the class City is the following:

```
add method increase_population
(amount:integer) in class City

body
increase_population(amount:integer) in
class City
co2 {(*self).population +=amount; }
```

Classes are related to each other by inheritance. Inheritance is a mechanism which allows the user to define classes in an incremental way by refining already existing ones. Normally in the definition of a class all information about the class structure is supposed to be known. So for example, the class City is a complete class because its type structure is well defined. However, when designing a schema the type associated to a class, i.e. the class definition, can be incomplete because non enough information about its structure was gathered.

We will show in the rest of the paper how to handle incomplete information when defining a class, thus allowing a powerful way to refine incomplete classes, called *generic*.

The inheritance mechanism of O_2 is based on the subtyping relationship, which is defined by a set inclusion semantics. O_2 allows multiple inheritance. Ambiguities arising from multiple inheritance are solved by forcing the user (designer) to explicitly redefine the attribute or method names when needed. If a class is created "with extension" then a named set value is created which will contain every object of the class and will persist.

O_2 allows object values to be manipulated by other methods other than those associated with the corresponding class. This feature is obtained by making "public" the type associated with the class. A detailed description of the O_2 data model can be found in [LeRi88,LeRi89a,LeRi89b], the description of

the system architecture is reported in [Vel89].

1.2 Incomplete Specifications

It is important to be able to define classes and objects even when the information about them is not totally specified. This is particularly important in the design phase. We should therefore be able to define a class which may have an incomplete associated type. Objects which belong to an incomplete class should be allowed as any other objects. We can distinguish two types of incomplete specifications:

a. *Incomplete schema.* This refers to the possibility of defining some classes (i.e. types and methods)in the schema which are associated with an incomplete type.

b. *Incomplete objects.* This refers to the possibility of expressing incomplete attribute values for objects.

1.3 Paper Organization

The paper is organized as follows: Section 2 will briefly introduce the notion of structural consistency in an object-oriented database. Section 3 will describe how to specify a partially defined schema by defining some classes which are associated to the "Generic" type. As a consequence we will have incomplete objects as instance of a generic class. Section 4 will describe how to specify incomplete attribute values for instance objects.

2 Structural Consistency

In this section we discuss a basic type of consistency that is relevant to the O_2 system (but in general to every object-oriented database system) , namely *structural* consistency.

Structural consistency refers to the static characteristics of the database. We informally recall here some of the fundamental characteristics of the O_2 system relevant to our discussion. Formal definitions of the O_2 system can be found in [LeRi89a,LeRi89b,LeRi88,Del89].

In the O_2 schema, classes are related with each other by inheritance; multiple inheritance is supported. Inheritance in O_2 is constrained by (pure) type compatibility; it is based on the notion of subtyping. Subtyping is a semantic relationship which connects two types. The semantics chosen for the subtyping relationships is set inclusion. That is:

Definition. A type T2 is a *subtype* of another type T1, denoted with T2$\leq$T1, if and only if every instance of T2 is also an instance of T1: $[\![\text{T2}]\!] \subseteq [\![\text{T1}]\!]$. Type T1 is called the supertype of T2.

Types can be complex; the subtyping relationship is defined for complex types as follows:

Definition. A *tuple* type t2 is a subtype of another tuple t1, denoted with t2 $\leq$ t1, if it is more defined: i.e. if it contains every attribute of its supertype, plus some new ones and/or refines the type of some attributes of its supertype.

Definition. A *set*-structured type set(T2) is a subtype of another set-structured type set(T1), denoted with set(T2) $\leq$ set(T1), if and only if T2 $\leq$T1.

The subtyping relationship is defined for classes as well:

Definition. A class C2 with associated type T2, is a *refinement* of another class C1(T1), denoted with C2 $\leq$ C1, if and only if the set of all possible objects which can belong to the extension of the class C2 is included into the set extension of C1. This definition requires T2 $\leq$ T1.

The operational part of O_2 is based on general purpose programming languages, such as C and Basic, enriched with language independent statements for database objects. Types also occur in methods' signatures.

Definition. The signature of a method is a pair denoted m:(w → t), where m is the name of the method, and w and t are type expressions. w stands for a sequence of type expressions where the first element is necessarily a class name.

Method signatures can be compared with each other with respect to their type expressions.

Definition. A method signature m2:(w2 → t2)is type-compatible with another method signature m1:(w1 → t1) if and only if the number of arguments in the two signatures is the same, and each element of the type expression of m2 (type or class) is a subtype (refinement) of the corresponding element of m1.

Attributes and methods are identified by name. Within the schema (attribute and method) names have a *scope rule.*

Definition. Given a (method or attribute) name n defined in a class C, the *scope rule* for n is the set of C and all subclasses of C (recursively obtained) where n is not locally redefined plus all classes where n is referred to with the "n from C" clause.

We are now ready to characterize the notion of structural consistency of a schema.

Definition. A schema is structurally consistent if and only if:

- its class structure is a direct acyclic graph (DAG);
- attribute types and method signatures related by inheritance in the schema are type-compatible.
- each name in the schema has a well-defined scope rule.

We consider a schema by definition structurally consistent.

Other object-oriented database systems offers slightly different data models. In particular, the main difference between O_2 and the Orion database system [Ba87a,Ba87b] is the existence in O_2 of *types* as well as classes. Types are not defined in Orion, and consequently the two data models are different.

Moreover, some features of Orion , such as default and shared attribute values, are not present in O_2; composite objects in Orion and O_2 are treated differently. In particular no notion of composite link attributes and dependent objects is present in O_2. GemStone [PeSt87] only supports simple inheritance, and therefore structural problems are simpler with respect to systems which allow multiple inheritance, such as Orion and O_2.

3 Generic Classes

The process of class definition should be as flexible as possible. In particular it should be possible to associate a class to a generic type and perhaps later in the design phase specialize it.

In order to provide such a facility we should give a precise answer to the following question:

a) What is the type associated to the system class OBJECT?

b) What is the semantics of a generic type?

In the O_2 system there are two separate structures: A class structure and a type structure. Classes have an associated type; types do not exist without a class. The top of the class structure is the system-defined class OBJECT. All classes in the class structure are by definition subclasses of the class OBJECT. The types associated to classes form a type structure. Types are related to each other by the subtyping relationships in accordance to the inheritance relationships in the class structure.

A type for the class OBJECT

Question a) says what is the type T in the type structure which corresponds to the system class OBJECT? Because OBJECT is a superclass of each class in the schema , we must have Ti $\leq$ T ,for each type Ti defined in the schema (and in general in the system). The relationships Ti $\leq$ T implies $[\![$ Ti $]\!] \subseteq [\![$T $]\!]$. The desired type T associated to the class OBJECT with the above property is the disjunction of types , i.e. T= $\vee$ each type. The disjunction of types is used in the language IQL [AbKa89]. Unlike IQL however, we propose to limit the use of the disjunction of types to the root element of the type structure, that is only to the type associated to the OBJECT class.

Semantics of the generic type

We want to allow the definition of a class without knowing the structure of its type. We call such a class a *generic* class. A generic type has a type associated , called *generic*, which corresponds to the empty type $\emptyset$ of IQL[AbKa89]. We follow the IQL approach and allow a value function to be a partial function as opposed to a total function.This corresponds to saying that the value of an object of type generic is not defined, denoted below with $\perp$.

Note the conceptual difference between a non-existent class and a generic class.

The use of generic classes allows the design of the schema to proceed in a step-wise refinement fashion, as the following example illustrates:

Example:

```
add class Monument with extension
          type  generic;

add method
Number_of_visitors(No_Visitors:integer)
in class Monument
body
Number_of_visitors(No_Visitors:integer)
in class Monument
     co2 {(printf("This is a test
message"), return a default integer)}
```

A generic class Monument has been created. A method Number-of- visitors has been associated to it. Note that in the body of the method the type associated to Monument is not used.

```
add class City
  type tuple (name:string;
              country:string;
              population:integer;
              monuments: set(Monument))
```

A class City is created that contains a complex attribute monuments which refers to the generic class Monument. The class City can be compiled without problem */

Objects for the above classes can be created without any problem: In particular objects of a generic class will have no value.

```
La_Scala =new(Monument)

Duomo =new(Monument)

Milano = new(City)
*Milano = tuple (name:Milano,
                 country:Italy,
                 population:2,000,000,
        monuments:(La_Scala, Duomo) )
```

The following table shows the object values and their associated type:

```
------------------------

 Class       Object_id
 Type        value
-------------------------

 Monument.    La_Scala
 generic      no-value

              Duomo
              no-value

City          Milano
tupl      (Milano,Italy,
            2,000,000,
          (La_Scala,Duomo))


----------------------
```

Because the generic type does not have value associated, it respects the inclusion semantics for subtyping. In particular we have: $\emptyset \leq \emptyset$, and $\emptyset \leq$ Ti (where Ti is a type in the system). Obviously, it does not hold that Ti $\leq \emptyset$ if Ti $\neq \emptyset$. Therefore, we can have a generic class in an inheritance hierarchy as a subclass of a well-defined class (i.e. with type not generic) but not viceversa as follows:

```
   add class Beautiful_Monument
       inherits Monument
       type generic;

   add class Nice_spot with extension
   type
   tuple(name:string,location:string);

   update Nice_spot superclass of
Monument;

   add class Perfect_City
      inherits City
      type tuple (monuments:set
of(Beautiful_monuments));
```

The resulting schema is the following:

```
City and Nice_spot
 subclasses of OBJECT,

Perfect_city
 subclass of City

Monument
 subclass of Nice_spot

Beautiful_monument
 subclass of Monument
```

Objects for such classes can be created as well:

```
Eifel_tower=new(Beautiful_monument)
Louvre=new(Beautiful_monument)
Paris=new(Perfect_city)
*Paris=type(Paris,France,3,000,000,
           {Eifel_Tower,Louvre})
```

Structural Consistency. The introduction of generic classes does not create problems when performing structural consistency check. In particular, the $\emptyset$ type corresponds to the bottom of the type lattice, and it is type-compatible with all types in the type structure.

The main reason in associating a class to a generic type is in delaying the specification of its type without affecting the definition of other classes (i.e. all classes can be compiled and methods can be executed). We should therefore be able to specialize the type generic when we wish. Specialization of the type of a class C to another type T is a type of schema update [Zic89].

We need to define a schema update: *Change the type of a class* which assigns a different type to a class. If the type is the same as the previous one the update does not have any effect. The extension of the class is modified in accordance with the new type. In particular, changing the $\emptyset$ type to a type T results in an *assignment* of a default value to the objects of the class extension.(The default value can be modified explicitly by the designer). For example, the update:

```
update Monument= type tuple
(name:string, year:integer)
```

This update results in a change to all objects associated to Monument without changing their object ids. In particular we have:

```
-----------------------
Class          Object_Id
Type           Value
-----------------------
Monument       La_Scala
(name,year)    (def.,def.)

               Duomo
               (def.,def.)
-----------------------
```

Note that changing the type of a class is different than changing the type structure of a class [Zic89]. When changing the type of a class C we need to check the type compatibility in all method signatures and attributes which use C. The update may create structural inconsistency and in this case must be refused.

The *generic* type could also be used in the method signature and attribute type.

```
type tuple (name:string,
other:generic)

method (C->generic)
```

Type-checking is still possible in this case. However, for the moment being we did not provide such a feature.

4 Incomplete Objects

There are two sources of incompleteness for an object:

- its associated class type is generic: there is no object value $\perp$.
- the class is not generic , but some of the object attribute values are not specified;

An example of the two kinds of incomplete object values is the following:

1. S.Ambrogio = new(Monument) - no value $\perp$
2. *La_ Scala = (LaScala, *unk*) - the second tuple attribute "year" is not known

In the second case we have that *unk* is a place- holder for any value from the *integer* domain. Modelling this kind of incomplete information is similar to that of handling null values in the relational data model [GZ88]. In the next subsections this analogy will be better explained.

Also the symbol $\perp$ must not be confused with the *NIL* object value often used in object-oriented systems to denote the object id of a non-existent object.

If one wants more specific refinement of the incomplete object attribute values, different types of null values can be introduced [Zic89]. This will be described in the next subsections.

4.1 Null Values in the Relational Data Model

Handling incomplete or missing information is traditionally achieved in relational database

theory with the introduction in the database of explicit "different" values, normally referred to as *null values*.

Depending on the degree of knowledge about the incompletness of the data one wants to model, we have several different proposals in the literature, ranging from simple ones to very sophisticated (and complicated) ones.

The basic idea behind the introduction of *null values* in a database is simply the necessity to express an exceptional behavior which is not contemplated in the definition of the database.

Consider a simple example, a company database which records name of employees together with their departments and phone numbers.

EMP	DEPT	TEL
Zhou	CS	5520
Smith	CS	null
Sechrest	CS	null
Black	null	null

There could be several situations which lead to an "exception" when filling the company database: A particular employee does not have a phone number, or we do not know his(her) number, we do not know whether he has a phone number, or we do not know whether he has more than one phone number, and so on. All these "exceptions" have in some way to be expressed in the database: The simpliest solution is to collect them under the general condition "something is not specified" and insert in the database a dummy value, say $\perp$, to express any of such anomalies. Even in this simplified case, we have to give a precise semantic to a relation which contains dummy values, and extend the relational operators to cope with this new value. Moreover, care must be taken when updating such database. If one wants to express additional knowledge about the incomplete data, different dummy values can be introduced in the database. We list some of the most studied types of null values: We have an *unk* (unknown) null value [Codd86] which tells us that the value exists for sure, but we do not know it. A *dne* (does not exist) null value [Vass79] which tells us that the value is not applicable (e.g. it does not exist). A *ni* (no information) null value [Zan84] which is a lower level placeholder for either *unk* or *dne* nulls, and which models all cases when not enough information is gathered to decide which of the previous nulls to use.

We have an *open* null value [GZ88] which leaves open the possibility that the attribute value may not exist, has exactly one unknown value, has several unkknown values.

With these additional values, we could fill the company database in a different way:

EMP	DEPT	TEL
Zhou	CS	5520
Smith	CS	unk
Sechrest	CS	dne
Black	unk	open

In the example, we have that Sechrest does not have a phone number, Smith has for sure one phone number, whose value is not known, and Black works in a department (but we do not know which one), and everything is left open for his phone number, i.e. he may have zero or one, or several phone numbers.

With the introduction of different *null values* we are able to express different sources of "incompletness" in our database. However,

the null values defined above, do not obviously model all possible situations of incompletness, for each of these situations it could be in theory possible to introduce some other dummy value. For example, in the literature *marked nulls* have been defined to tell that a particular unknown value is equal (or different) to another unknown value. The limit in introducing new symbols is the complexity in defining a "reasonable" yet "simple" semantics for a relation with all these null values and for the relational operators.

Maier [Maier85] defined some criterias to decide whether the semantics of relational operators in presence of null values are "reasonable" in terms of properties such as *adequacy*, *preciseness*, and *faithfulness*.

Informally we can say that introducing different null values in a database is a way to make the database itself more closed to the real life application it models.

4.2 Null Values in the Nested Relational Model

The Nested Relational Data Model (NRDM) is a data model which extends the calssical relational model to allow attributes of a ralation to be relation themselves, thus modeling hierachical structures, but still keeping the relational approach [ScSc86,ThFi86]. We do not address here the question whether this is the data model one wants to use to model complex structures. We want to point out an interesting problem with this model which also exists in the object-oriented data model (see section 4.4).

Because in the NRDM an attribute can be a relation as well, that is a set of tuples, the domain of a complex attribute includes the empty set as legal value.

This is shown with the help of this simple nested database:

EMP	DEPT	{TEL}
Victor	EE	{6640,6645,6648}
O'Brian	EE	{ }

The second tuple of the nested relation tells us that employee O'Brian works in the Eletrical Engineering department an has an "empty set" of telephone numbers. Now the key of the game is that the empty set already tells us of an exceptional case: No phone numbers have been given to O'Brian. The claim here is that the empty set is a particular type of null value. The difference, with respect to the case of the standard (without nulls) flat relation is that in the NRDM the definition of the empty set is a legal instance of an attribute, while in the (flat) relational model there does not correspond any legal value for it.

This assimmetry gets obvious when one wants to "flatten" a nested relation which contains an empty set. If and only if we allow an explicit *dne* value in the relational model to exist (and therefore extending the domain of the attributes), then the operator of *UNNEST(TEL)* applied to the nested relation above and which "flattens" the set-valued "TEL" attribute, results in following relation without losing information:

EMPT	DEPT	TEL
Victor	EE	6640
Victor	EE	6645
Victor	EE	6648
O'Brian	EE	dne

That is, in a world with explict null values, we can consider the empty set semantically the

correspondent to the *dne* null value for single-valued attributes. The empty set is then considered an explicit null value. This approach can be generalized with the introduction of the other explict null values in the NRDM.

An example of a nested relation with explict null values is the following:

EMP	DEPT	{TEL}
Victor	EE	{unk, unk, umk}
O'Brian	EE	{ }
Black	open	{5519}

A similar argumentation works for the extensions to the NRDM model which allows lists and multisets, besides sets, for complex attributes [Dad86, Cer89, Gu89].

4.3 Closed vs. Opened world Databases

There are two possible ways to look at a database: Considering the database the repository of all our knowledge about the universe: This informally corresponds to the *closed world* assumption; or considering the database the repository only those facts we can prove as true: This informally corresponds to the *open world* assumption [Reit78].

The introduction of null values determines the way a database is considered: No null values or only *dne* and/or, *unk* nulls in a relation corresponds to a *closed world* database; only *ni*, and/or *open* nulls corresponds to a *open world* database [GZ88].

4.4 How Does Object Orientation Cope with Incomplete Objects?

The way an object-oriented database at the moment handles incomplete information looks very much the same as in a closed world nested relational database without explicit null values. We try to better explain this concept with some examples.

Suppose we have defined in our system the persistent class propeller, from [Banc88]:

```
add class Propeller with extension
type tuple (fuel:string,
            power:integer,

nominal_consumption:float,
            identifier:string)

add method
  Consumption(var fuel:float)
  body { fuel =
self.nominal_consumption }
```

What happens if we do not know the intial values of the instance variables of an object of class Propeller? Two cases are possible, "crude" default values are given, or a runtime error occurs during execution of a method which refers to a non initialized variable. Using explicit null values is the natural way to 'fill in' object values when not totally specified.

Consider the followng classes which define a complex structure as follows:

```
add class Part
type tuple (name:string,
            specific_weight:float)

add class Vehicle
type tuple (name:string,
            propeller:Propeller,
            designer:string,
            components: set(Part) )
```

The class vehicle has an attribute, components, which is a set of objects of class Part.

What happens if attribute componets has an empty set value? The situation resembles that of the NRDM: The empty set is a legal instance value for the type set. We suggest that the empty set must be considered an explicit null value. If an object of the class Vehicle does not have any subcomponents then we write *empty set* for the attribute value components. The assimmetry in the treatment of empty sets in the object-oriented model is similar to that of the NRDM model: No legal instance value exists for expressing the same situation for *basic* or *tuple* attribute types. Introucing explicit null values for such attributes solves the problem as in the NRDM.

When allowing null values in the object-oriented data model, care must be taken in defining the semantics of methods. Every time the designer defines a new method, he/she should contemplate the possibility of null values and define the semantics accordingly.

The introduction of explicit null values should also be considered if objects are related to each other by inheritance. A possible way to solve the problem is to impose an order relation on null values which defines the quantity of information each particular null value represents. A similar approach is followed in the classical relational model to compare tuple with null values [Zan84].

In this paper we do not address this issue any further.

5 Conclusions and Future Work

The introduction of *generic* classes helps incremental design. We have shown how generic classes fit nicely in the O_2 typing system, and in general in any object oriented database system which provides inheritance.

Incomplete information can be expressed in an object instance with the use of explicit null values, in a similar way as in the relational and nested relations models. We have also shown some interesting analogies in the way the NRDM and the object oriented model handle empty sets.

We conlude by saying how the introduction of concepts such as generic classes and null object values poses some interesting questions in the theory of object oriented databases which seem interesting to address. More work is needed to render our approach operational.

Acknowledgements

Some of the ideas presented in this paper emerged after discussions with Serge Abiteboul and Paris Kanellakis. In particular the concept of generic class has been influenced by the IQL language [AbKa89].

The members of the Altair group and in particular Francois Bancilhon, Christine Delcourt,Claude Delobel, Sophie Gamerman, Christophe Lecluse, Mark James, Philippe Richard, and Fernando Velez gave me useful explanations on the O_2 system.

6 References

[ACMTOOIS87] ACM Transactions on Office Information Systems, Special Issue on Object-Oriented Systems, Vol.5, No. 1, january 1987.

[AtBu87] Atkinson M.P., Buneman O.P., "Types and Persitence in Database Programming Languages", ACM Computing Surveys, Vol.19, No.2, June 1987.

[Ba87a] .J. Banerjee et al,"Semantics and Implementation of Schema Evolution in Object-Oriented Databases", ACM SIGMOD 1987.

[Ba87b.] J. Banerjee et al., "Data Model Issues for Object-Oriented Applications", ACM TOOIS, vol.5,No.1, January 1987.

[Banc88] Bancilhon et al., "The Design and Implementation of O2 an Object Oriented Database System", Altair Report 20-88, 1988.

[Ca88] Cardelli L., "A Semantics of Multiple Inheritance", Information and Control 76, 1988, Academic Press.

[Cer89] Ceri, S., Crespi-Reghizzi S., Lamperti G., Lavazza L., Zicari R., ALGRES: An Advanced Database System for Complex Aplications", IEEE SOFTWARE to appear.

[Codd86] Codd E.F., "MIssing Information in Relational Databases", SIGMOD Record, Vol.15, No.4, December 1986.

[DaTo88] Danforth S, Tomlinson C., "Types Theoties and Object-Oriented Programing", ACM Computing Surveys, Vol.20, No.1, March 1988.

[Gu89] Gueting R., Zicari R., Choy D., "Am Algebra for Structured Office Documents", ACM Transactions on Office Information Systems, to appear.

[Maier83] Maier D., "The Theory of Relational Databases", Computer Science Press, Rockville MD, 1983, Chapter 12.

[OOPSLA87-88] Proceedings Object Oriented Programming, Systems, Languages and Applications, Florida 1987, San Diego 1988.

[PeSt87] D.J. Penney, J. Stein, "Class Modification in the GemStone Object-Oriented DBMS", ACM OOPSLA October 1987.

[Reit78] Reiter R., On Closed World Databases", in Logic and Databases, (H. Gallaire, J. Minker, J.M. Nicolas eds.), PLenum Press, New York, 1978.

[LeRi89a] C. Lecluse,P. Richard, "The O_2 Database Programming Language", Altair Report 26-89, January 1989 , in Proc. VLDB 89 , Amsterdam, August 1989.

[Vel89] F. Velez et al.,"The O_2 Object Manager: an Overview", Altair, February 1989, in Proc. VLDB 89, Amsterdam , August 1989.

[GZ88] G. Gottlob, R. Zicari, "Closed World Databases Opened through Null Values", in Proc. 14th VLDB Conference, Los Angeles, August 1988.

[AbKa89]
S. Abiteboul, P. Kanellakis,"Object Identity as a Query Language Primitive", Proc. ACM SIGMOD, Oregon, June 1989.

[LeRi89b] C. Lecluse, P.Richard, "Modeling Complex Structures in Object-Oriented Databases", in Proc. ACM PODS, 1989.

[Del89] C. Delobel, "A Formal Framework for O2 data model", Altair report, to appear.

[Dad86] Dadam P., et al. "A DBMS to Support Extended NF2 Relations:An Integrated View on Falt Tables and Hierarchies", Proc. ACM Sigmod, 1986, Washington.

[LeRi88] C. Lecluse, P. Richard,"Modeling Inheritance and Genericity in Object-Oriented Databases", in Proc. ICDT-88, Bruges, August 1988.

[ScSc86] Schek H., Scholl M., "An Algebra for the Relational Model with Relation Valued Attributes", Information Systems, vol.11, No.2, 1986.

[ThFi] Thomas S.J., Fisher P., "Nested Relational Structures", in Advances in Computing Research, Vol.13, JAI Press, 1986.

[Vass79] Vassiliou Y., "Null Values in Database Management Systems: A Denotational Semantics Approach", Proc. ACM SIGMOD, Boston 1979.

[Zan84] Zaniolo C., "Database Relations with Null Values", Journal of Computer and System Sciences, 28, 1984.

[Zic89] R. Zicari, "A Framework for Schema Updates in O_2", report No. 89-036, Politecnico di Milano, Milano, Italy and GIP Altair, France.