

On Formal Models for Object-Oriented Databases

Gottfried Vossen

Fachbereich Mathematik, Arbeitsgruppe Informatik
Justus-Liebig-Universität Gießen
Arndtstraße 2
D-6300 Gießen, Germany

March 1992

Abstract

While object-oriented database management systems are already arriving in the marketplace, their formal foundations are still under development. In this paper, one central aspect of such foundations, *formal models* for object-oriented databases, is considered. It is discussed why a formal model is desirable, what it is supposed to comprise in this context (a structural as well as a behavioral part), and how this can be achieved; to this end, the central ingredients which are shared by many proposed models are presented in some detail. This carries over to design issues for database descriptions in an object-oriented model, for which two distinct strategies are outlined. Finally, the question is discussed whether the modeling concepts described are indeed the ones that the applications which originally triggered the merger of database technology with object-oriented concepts need. Our argument is that this is only partially the case, and two promising directions for future work are sketched.

1 Introduction

Object-oriented data models represent a current end-point in the evolution of data models [24], the central members of which are summarized in Figure 1. This paper discusses various aspects of *formal models* for object-oriented databases: *why* they are desirable, *what* they have to comprise, *how* that can be achieved, and *which* models seem to be adequate. On the side, it also discusses design questions for object-oriented database schemata.

Object-orientation as a paradigm is based on the following five fundamental principles [6]:

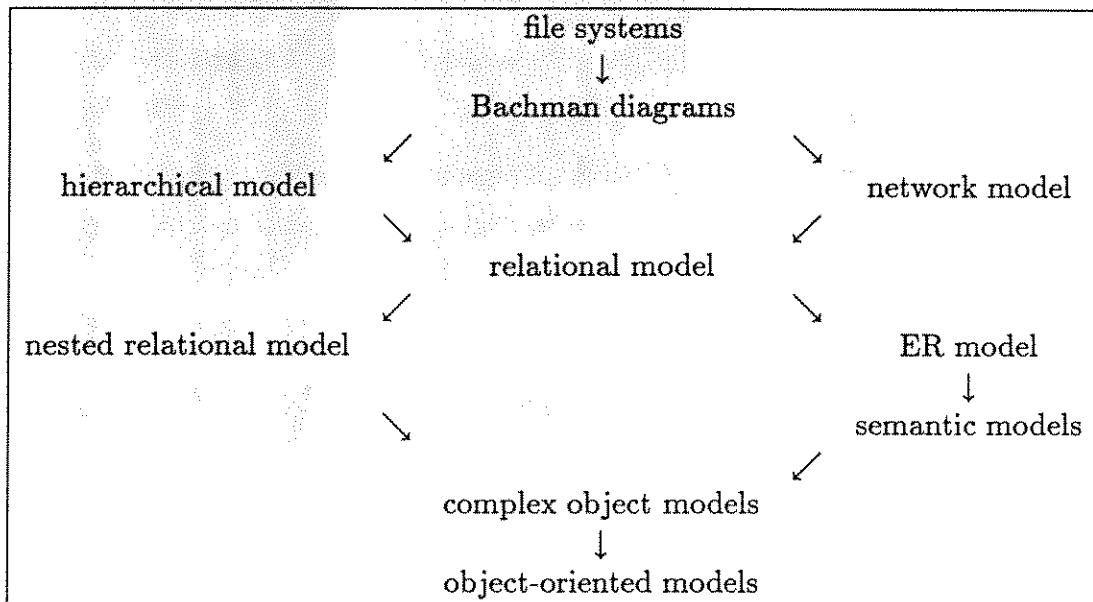


Figure 1: Evolution of Data Models.

1. Each entity of the real world is modeled as an *object* which has an existence of its own, manifested in terms of a unique *identifier* distinct from its value;
2. each object has *encapsulated* into it a *structure* and a *behavior*; the former is described in terms of attributes (*instance variables*), where attribute values, which together represent the state of the object, can be other objects so that complex objects can be defined via aggregation; the latter consists of a set of *methods*, i.e., procedures that can be executed on the object;
3. the state of an object can be accessed or modified exclusively by sending *messages* to the object, which cause it to invoke corresponding methods;
4. objects sharing the same structure and behavior are grouped into *classes*, where a class represents a “template” for a set of similar objects; each object is an instance of some class;
5. a class can be defined as a specialization of one or more other classes; a class defined as a specialization forms a *subclass* and *inherits* both structure and behavior (i.e., attributes and methods) from its *superclasses*.

In databases these principles have to be augmented with additional requirements, namely the support of *complex objects*, i.e., highly structured information, and type-system *orthogonality* and *extensibility*; it should be possible to define new types at any time by applying given constructors to already defined types in a vastly arbitrary fashion.

Consequently, an object-oriented *data model* has to capture this entire list of requirements, and must thus intuitively be more complex than any previously developed model. However, many system developers seem not to care about *formal models* as a solid foundation of their system, but simply design a “data definition language” in which the features listed above can be coded. In our opinion, a formal model for object-oriented databases basically has to capture the same intuitions as models for other types of databases, which are the following:

1. *It has to provide an adequate linguistic abstraction for certain database applications.* For example, it should abstract from the physical organization of the database in question and hence support *physical data independence*.
2. *It should provide a precise semantics for a data definition language.* For example, the relational language SQL comprises a statement for creating tables in a relational database, but it is the *relational model* underlying SQL which yields a semantics for that statement.
3. *It has to be composed of both a specification and an operational part.* In the case of the relational model, the former refers to tables or relations as the structuring mechanism, and the latter to, say, the operations of relational algebra. In this paper, we concentrate on the specification component of object-oriented models, since this appears already complex enough.
4. *It represents a computational paradigm as a basis for formal investigations.* For example, the relational model has proved to be an elegant, well-motivated and appropriately restricted such paradigm for database management, and allowed, among other things, for in-depth investigations of database design or query languages. We see no reason why the same should not hold for object-oriented models. However, the situation here is obviously more complicated, due to the requirement to model both structure and behavior, but as will be seen below, a number of corresponding investigations are already underway.

In this paper, we do not present a comprehensive survey of formal models for object-oriented database which have been proposed in the literature, but instead try to point out some fundamental techniques of how such models can be obtained. The result can be considered as a framework in which the essentials of the object-oriented paradigm can be expressed concisely and even further studied. Indeed, we will indicate how several investigations considering the behavioral aspects fit into this framework.

As an aside, we will also discuss the question of how database design can be carried out for such models. Traditionally, a design process distinguishes a *conceptual* from a *logical* design phase, where the goal of the former is to model an application in a target-model independent fashion, and that of the latter is to transform the result of conceptual modeling into the given target model. We question this distinction

for object-oriented databases on the basis of the observation that each database description in a model supporting aggregation and inheritance has a natural graphical representation, which can even be constructed in the first place.

Finally, we try to answer the question whether the current situation with formal models for objects and object-oriented databases is indeed what the new applications like CAD or CASE, which originally triggered the development of such databases, really need. We argue that this is not the case, give reasons to support this claim, and sketch promising directions for future research.

The organization of the paper is as follows: In Section 2 we survey the requirements which originally motivated the departure from relational databases as well as the merger of databases with object-orientation. The core requirements, the modeling of highly structured information and that of behavior, are met by a number of data models that have recently been proposed, and we describe a framework in which most of them can uniformly be cast. In Section 3, we discuss design issues for object-oriented databases along the lines indicated above. In Section 4 we present our arguments that new types of models are what nontraditional, in particular technical applications of databases really need, and in Section 5 we draw some conclusions.

2 What is Needed: A Formal Framework

The current situation in the field of object-oriented database systems (OODBS) is remarkably different from that of relational systems some 20 years ago, since the latter started from a strong theoretical foundation, but no consensus is yet in sight regarding a formal theory of OODBS; despite of this, several OODBS are already commercially available, and many others close to being marketed.

Fortunately, a formal theory of OODBS seems well-underway, since a number of issues are already under in-depth investigation, or at least formalizations have been proposed which point in the right direction, i.e., allow for new types of formal studies. In this section, we briefly describe these directions as they pertain to formal models for OODBS, and we do so by distinguishing structural from behavioral aspects. Thus, we generally consider *schemas*, the central notion of a conceptual database description, to be pairs of the form

$$S = (S_{\text{struc}}, S_{\text{behav}});$$

we now consider each component in turn and also indicate how they might interact.

We mention, however, that while it is generally agreed that an object-oriented data model has to capture both structure and behavior, the former can be obtained by using the experience from the relational, nested relational and complex-object models, but the latter represents a completely new challenge to database researchers. Consequently, a consensus seems currently achieved for structure, but not for behavior.

2.1 Modeling Structure

Regarding the modeling of structure, more precisely highly-structured information, complex data types are all that is basically needed, since they serve as descriptions for domains of complex values. One way to introduce such types, i.e., to define a *type system* T , is the following:

- (i) integer, string, float, boolean $\subseteq T$;
- (ii) if A_i are distinct attributes and $t_i \in T$, $1 \leq i \leq n$, then $[A_1 : t_1, \dots, A_n : t_n] \in T$ (“tuple type”);
- (iii) if $t \in T$, then $\{t\} \in T$ (“set type”);
- (iv) if $t \in T$, then $\langle t \rangle \in T$ (“list type”).

In other words, a type system is made up of *base types*, from which complex types may be derived using (eventually attributes and) *constructors*. Note that this requires nothing additional but the availability of attribute names. Clearly, other base types as well as additional or alternative constructors could straightforwardly be included.

Example 1 The above form of a type system already allows for definitions like the following, which is intended to describe a data type for persons:

```
[ name: [ first: string, last: string ],
  age: integer, married: boolean,
  address: [ street: string, city: string, zip: integer ],
  children: < string >,
  works_for: { [ company: [ name: string,
                          location: string ],
                percentage: integer ] } ]
```

This is supposed to model a data type for persons who have a name, an age, a marital status, an address, a list of children, and a set of part-time jobs. \square

The notion of a domain as a “reservoir” of possible values can be defined as follows; it just has to obey constructor applications:

- (a) $\text{dom}(\text{integer})$ is the set of all integers; dom is analogously defined for string, float, boolean;
- (b) $\text{dom}([A_1 : t_1, \dots, A_n : t_n]) := \{[A_1 : v_1, \dots, A_n : v_n] \mid (\forall i, 1 \leq i \leq n) v_i \in \text{dom}(t_i)\}$;
- (c) $\text{dom}(\{t\}) := \{\{v_1, \dots, v_n\} \mid (\forall i, 1 \leq i \leq n) v_i \in \text{dom}(t)\}$;
- (d) $\text{dom}(\langle t \rangle) := \{\langle v_1, \dots, v_n \rangle \mid (\forall i, 1 \leq i \leq n) v_i \in \text{dom}(t)\}$.

In a structurally *object-oriented* context, the first thing that needs to be introduced beyond complex types and domains as defined above is the possibility to *share* pieces of information between distinct types, or to *aggregate* objects from simpler ones. At the level of type declarations, an easy way to model this is the introduction of another reservoir of names, this time called *class names*, which are additionally allowed as types. In other words, *object types* are complex types as above with the following new condition:

- (v) $C \subseteq T$, where C is a finite set of class names.

This states nothing but the fact that class *names* are allowed as types (below we will complement this with the requirement that classes themselves *have* types).

The intuition behind this new condition is that objects from the underlying application all are distinguished by their identity, get collected into classes, and can reference other objects (share subobjects). To provide for this at the level of domains, let us first assume the availability of a finite set *OID* of *object identifiers* which includes the special identifier *nil* (to capture “empty” references); next, *object domains*, i.e., sets of possible values for objects are complex values as above with the following additional condition:

- (e) $\text{dom}(c) = \text{OID}$ for each $c \in C$.

Thus, classes are assumed to be instantiated by objects (class-name types take object identifiers as values, in the same way as, say, the *int* type takes integer numbers as values).

Example 2 Returning to the above example, personal data can now be defined as follows: Let $C = \{ \textit{Person}, \textit{Address}, \textit{Company} \}$. Since both persons and companies have an address, and since children are in turn persons, types can be associated with these class names in the following way:

1. Type of class *Person*:

```
[ name: [ first: string, last: string ],
  age: integer, married: boolean,
  address: Address,
  children: < Person >,
  works_for: { [ company: Company, percentage: integer ] } ]
```

2. Type of class *Address*:

```
[ street: string, city: string, zip: integer ]
```

3. Type of class *Company*:

```
[ name: string, location: Address ]
```

Domains for these types then provide object identifiers wherever classes are referenced; nil would be used to express that children might not yet work. \square

As was mentioned in the Introduction, the object-oriented paradigm has another dimension for organizing information besides aggregation, which is inheritance, or the possibility to define a class as a specialization of one or more other classes. To this end, a *subtyping relation* is needed through which it can be expressed that a subclass *inherits* the structure of a superclass. Such a relation can be defined in various ways; one is the following [29]:

Let T be a set of object types. A subtyping relation $\leq \subseteq T \times T$ is defined as follows:

- (i) $t \leq t$ for each $t \in T$,
- (ii) $[A_1 : t_1, \dots, A_n : t_n] \leq [A'_1 : t'_1, \dots, A'_m : t'_m]$ if
 - (a) $(\forall A'_j, 1 \leq j \leq m)(\exists A_i, 1 \leq i \leq n) A_i = A'_j \wedge t_i \leq t'_j$,
 - (b) $n \geq m$,
- (iii) $\{t\} \leq \{t'\}$ if $t \leq t'$,
- (iv) $\langle t \rangle \leq \langle t' \rangle$ if $t \leq t'$.

With these preparations, we arrive at the following definition for objectbase schemas that can describe structure of arbitrary complexity: A *structural schema* is a named quadruple of the form

$$S_{\text{struc}} = (C, T, \text{type}, \text{isa})$$

where

- (i) C is a (finite) set of class names,
- (ii) T is a (finite) set of types which uses as class names only elements from C ,
- (iii) $\text{type} : C \rightarrow T$ is a total function associating a type with each class name,
- (iv) $\text{isa} \subseteq C \times C$ is a partial order on C which is consistent w.r.t. subtyping, i.e., $c \text{ isa } c' \Rightarrow \text{type}(c) \leq \text{type}(c')$ for all $c, c' \in C$.

This definition resembles what can be found in a variety of models proposed in the literature, including [16,18,19,26,29] and others. Notice that it still leaves several aspects open, like single vs. multiple inheritance; if the latter is desired, a condition needs to be added stating how to conflicts should be resolved. Also, implementations typically add a number of additional features not included above, like

- attributes as functions [23,30],

- a distinction of class attributes from instance attributes (the latter a shared by all objects associated with a class, while the former represent, for example, aggregate information like an average salary only relevant to the class as a whole) [5],
- a unique root of the class hierarchy from which every class inherits [19],
- a distinction between private and public attributes [10],
- a different set of constructors (like one with an additional array constructor to describe matrices),
- an explicit inclusion of distinct types of relationships between classes and their objects (in particular various forms of composition, see [17]),
- integrity constraints which represent semantic information on the set of valid databases instances (a proposal in that direction appears in [3,4], where *object constraints*, *class constraints*, and *database constraints* are distinguished).

2.2 Modeling Behavior

The second important aspect of an object-oriented database is that it is intended to capture behavior, besides structure. To this end, the relevant intuition is that classes have attached to them a set of messages, which are specified in the schema via signatures, and which are implemented as methods. In addition, behavior can be inherited by subclasses, and message names can be overloaded, i.e., re-used in various contexts.

So a *behavioral schema* is a named five-tuple of the form

$$S_{\text{behav}} = (C, M, P, \text{messg}, \text{impl})$$

where

- (i) C is a (finite) set of class names as above (again needed here since references to it have to be made),
- (ii) M is a (finite) set of *message names*, where each $m \in M$ has associated with it a nonempty set $\text{sign}(m) = \{s_1, \dots, s_l\}$, $l \geq 1$, of signatures; each s_h , $1 \leq h \leq l$, has the form

$$s_h : c \times t_1 \times \dots \times t_p \rightarrow t$$

for $c \in C$, $t_1, \dots, t_p, t \in T$ (each signature has the receiver of the message as its first component),

- (iii) P is a (finite) set of methods or programs,

- (iv) $\text{messg} : C \rightarrow 2^M$ s.t.
 $(\forall c \in C) (\forall m \in \text{messg}(c)) (\exists s \in \text{sign}(m)) s[1] = c$
- (v) $\text{impl} : \{(m, c) \mid m \in \text{messg}(c)\} \rightarrow P$ is a partial function.

In combining structural and behavioral schemas, we finally obtain the following: An *objectbase schema* has the form

$$S = (C, (T, \text{type}, \text{isa}), (M, P, \text{isa}, \text{messg}, \text{impl}))$$

S is called *consistent* if the following conditions are satisfied:

- (i) $c \text{ isa } c' \Rightarrow \text{messg}(c') \subseteq \text{messg}(c)$ for all $c, c' \in C$,
- (ii) if $c \text{ isa } c'$ and $s, s' \in \text{sign}(m)$ for $m \in M$ such that $s : c \times t_1 \times \dots \times t_n \rightarrow t$, $s' : c' \times t'_1 \times \dots \times t'_n \rightarrow t'$, then $t_i \leq t'_i$ for each i , $1 \leq i \leq n$, and $t \leq t'$,
- (iii) $(\forall m \in \text{messg}(c)) (\exists c' \in C) c \text{ isa } c' \wedge \text{impl}(m, c')$ is defined.

Condition (i) just says that subclasses inherit the behavior of their superclasses. Condition (ii) says that message-name overloading is done with compatible signatures (called the *covariant condition* in [19]). Finally, Condition (iii) states that for each message associated with a class, its implementation must at least be available in some superclass.

It is interesting to note that various natural conditions can be imposed on the programs that are used as implementations of messages. We now sketch one of them, which is based on the view that programs are functions on domains [19]. More formally, if $m \in M$ and $s : c \times t_1 \times \dots \times t_n \rightarrow t \in \text{sign}(m)$, then $\text{impl}(m, c)$, if defined, is a program $p \in P$ of the form

$$p : \text{dom}(c) \times \text{dom}(t_1) \times \dots \times \text{dom}(t_n) \rightarrow \text{dom}(t)$$

The condition in question informally states that if message overloading appears in isa-related classes (so that the corresponding signatures satisfy the covariant condition), then the associated programs coincide (as functions) on the subclass. More formally, we have: If $|\text{sign}(m)| > 1$ for some $m \in M$, then the following holds: If $s, s' \in \text{sign}(m)$ such that

- $s : c \times t_1 \times \dots \times t_n \rightarrow t$,
- $s' : c' \times t'_1 \times \dots \times t'_n \rightarrow t'$,
- $c \text{ isa } c'$,
- $t_i \leq t'_i$ for each i , $1 \leq i \leq n$, $t \leq t'$, and
- $\text{impl}(m, c) = p$, $\text{impl}(m, c') = p'$,

then p and p' agree on $\text{dom}(c) \times \text{dom}(t_1) \times \dots \times \text{dom}(t_n)$.

A variety of formal investigations for behavioral schemata in the sense defined above can already be found in the literature, which investigate questions including the following:

- termination of method executions,
- limited depth of method-call nestings (an issue related to precompilation of method executions),
- well-definedness of method calls, i.e., consistency as well as reachability considerations (issues related to type inference and schema evolution),
- expressiveness of method implementation languages (relative to some notion of completeness),
- complexity of method executions,
- potential parallelism of method evaluations.

To investigate such issues, our general notion of schema is made precise in various ways. For example, [14] fixes a simple imperative language for implementing methods as *retrieval programs*, contrasts them with *update programs* and shows undecidability results for the latter. [1,2] as well as [9] introduce distinct notions of a *method schema* to study behavioral issues of OODBS; for example, [2] investigates implications of the covariance condition using the formalism of program schemas, while [9] looks at tractability guarantees corresponding to those known for relational query languages. The interested reader is referred to the cited references for more information.

We conclude this section with a brief indication of how *object databases*, i.e., sets of class instances or extensions, can be defined over a given schema: For a given objectbase schema S , an *objectbase* over S is a triple $d(S) = (O, \text{inst}, \text{val})$ s.t.

- (i) $O \subseteq \text{OID}$ is a finite set of object identifiers,
- (ii) $\text{inst}: C \rightarrow 2^O$ is a total function satisfying the following conditions:
 - (a) if $c, c' \in C$ are not (direct or indirect) subclasses of each other, then $\text{inst}(c) \cap \text{inst}(c') = \emptyset$,
 - (b) if $c \text{ isa } c'$, then $\text{inst}(c) \subseteq \text{inst}(c')$,
- (iii) $\text{val}: O \rightarrow V$ is a function s.t. $(\forall c \in C) (\forall o \in \text{inst}(c)) \text{val}(o) \in \text{dom}(\text{type}(c))$.

Figure 2 summarizes the central aspects of the object model we have described.

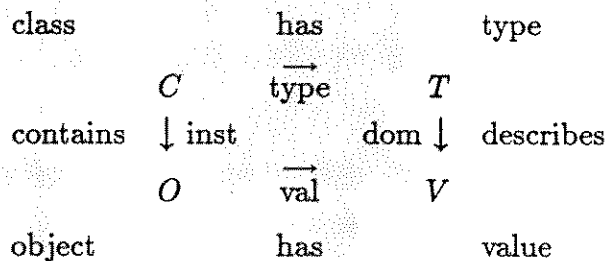


Figure 2: The central ingredients of an object model.

3 Design Issues

In this section we briefly discuss an aspect of database design in the object-oriented context which seems to arise as soon as aggregation and specialization, core data modeling features in general, are directly available in a target model.

Traditionally, a distinction is made in every database design process between the *conceptual* and the *logical* design phase. The goal of the former is to obtain an abstract representation of the application in question, using some high-level model capable of supporting classification, aggregation and specialization in some way. In general use today is Chen's *Entity-Relationship* (ER) model, for which even design *methodologies* (i.e., prescriptions of how to employ the model) are readily available. The goal of logical design is to take the target model of the system to be used into account, and to transform the result of the conceptual design phase into that model. For example, if the target model is the relational one and conceptual design has resulted in an ER diagram, rules can be applied that state how to obtain relational schemata from the components of that diagram.

For modeling complex structures or highly structured information, semantic models have been proposed that go beyond the modeling capabilities of the ER model. On the other hand, the distinction between conceptual and logical design, now with different models at each phase, still makes sense, since commercial systems do not support semantic models directly. For example, [21] describes a design tool in which conceptual design is based on the GSM model from [13], and where logical design transforms GSM diagrams into class declarations for the O_2 system [10].

While the distinction of these two phases is well-motivated in contexts where the target model is restricted with respect to its modeling capabilities when compared to a semantic model (including ER), so that database design can initially concentrate on the application instead of on the system, we believe that the two phases of conceptual and logical design can be integrated with an object-oriented model. This perception is based on the following observation: Consider a structural schema of the form $S = (C, T, \text{type}, \text{isa})$. With this schema a labelled *structure graph* $G(S) = (V, E, l)$ can be associated as follows:

- (i) $V = C$, i.e., the class names are used as nodes,
- (ii) an edge (c, c') is in E if one of the following conditions is satisfied:
 - (a) either c isa c' ; in this case the edge is labelled "isa",
 - (b) or an attribute from the type of c has c' as its domain type, i.e., $\text{type}(c)$ contains a declaration of the form " $A : c'$ "; in this case the edge gets labelled by A .

Thus, edges of the former type capture specialization, while edges of the latter type capture aggregation.

Example 3 Consider the following set of class declarations in the language of the O_2 system [10]:

```

add class PlaceToGo
  type tuple(name : string,
             picture : Bitmap,
             address : Address,
             thingsToDo : set(ThingToDo));

add class ThingToDo
  type tuple(name : string,
             place : PlaceToGo);

add class Address
  type tuple(number : integer,
             street : string,
             city : City,
             postalCode: string);

add class City
  type tuple(name : string,
             country : string,
             monuments : set(Monument),
             hotels : list(Hotel));

add class Hotel
  type tuple(name : string,
             address : Address,
             stars : integer,
             facilities : list(string));

```

```

add class Monument
  inherits PlaceToGo
  type tuple(buildingDate : Date,
             closingDays : list(string),
             admissionFee : integer,
             architect : Person);

add class ScheduledEvent
  inherits ThingToDo
  type tuple(schedule : set(Date));

add class Conference
  inherits ScheduledEvent
  type tuple(speaker : Person,
             topic : string);

```

Figure 3 shows its structure graph, where classes are represented by rectangles, and single [double] arrows indicate aggregation [specialization], resp. (and labels are not shown). □

Clearly, the graphical representation of a structure graph can be refined in such a way that the attributes appearing in the type declaration of each class (at least those appearing at the highest level of nesting) are shown individually, so that aggregation edges can be depicted in greater detail.

A natural question to ask now is whether this can be reversed. In other words, we can imagine a model for conceptual design capable of producing descriptions as in Figure 3, so that logical design, i.e., the transformation of such a diagram into an object-oriented target model is “built-in”. Indeed, the GOOD model described in [11,12] can be considered as an approach in this direction, and a further investigation of this observation appears feasible.

4 What is Wanted

We now turn to the final issue related to formal models we want to discuss, namely the question of whether models defined along the lines of what we did in Section 2 are indeed appropriate for nonstandard applications of databases. This question is natural to be asked, since it occurs to us that although applications like CAD or CASE are always mentioned as those which require data models beyond relational capabilities, examples illustrating new models are always phrased in terms of well-known “supplier-parts”, “university” or other standard applications. While it is certainly true that even traditional applications can benefit from the object-oriented paradigm, we feel that this discrepancy is not a coincidence, and try to argue in this section why this is so. As a result, the answer to the question of whether what has been proposed so far is appropriate will be negative.

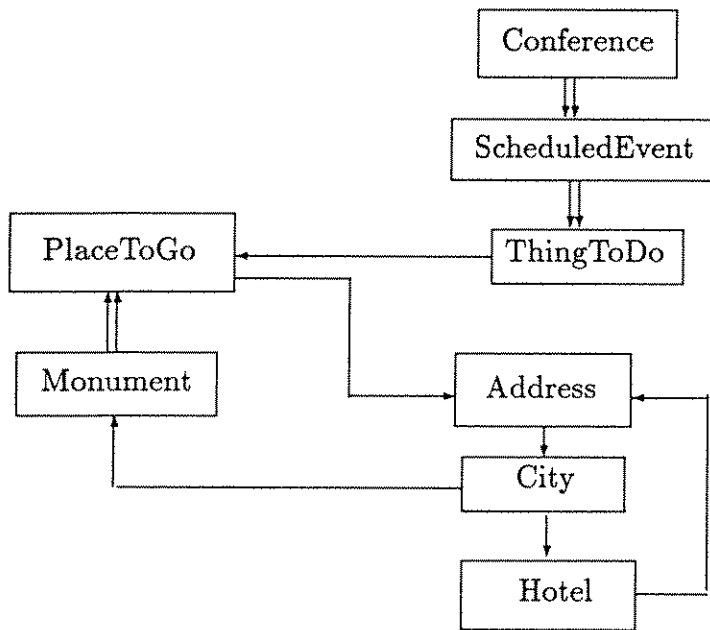


Figure 3: Graphical representation of the O_2 schema declaration.

Open Modeling Issues

We next survey several modeling issues in object-oriented databases which have not yet received enough attention:

1. *Entities can have roles that vary over time.* For example, some person object may at one point be a student, at another an employee, and at a third a club member; while the person's identity never changes, its type changes several times.
2. *Entities can have multiple types at the same time.* For example, a person may be a student, an employee, and a club member simultaneously. So far the only way to represent this in an object-oriented database is by multiple inheritance, but this might not be appropriate since it can result in a combinatorial explosion of sparsely populated classes [22].
3. *Objects can be in various stages of development.* For example, in a design environment it is usually necessary to maintain incomplete designs, i.e., objects whose types get completed in the course of time.
4. *Classes may contain "too few" instances.* For example, consider a database in which all persons living in a large country are represented. In this context, so many combinations of meaningful properties have to be distinguished that it might become necessary to introduce artificial name constructions for classes, like *unmarried-nonstudent-autoOwner-renter-taxpayer* [27], and each such class has only very few instances. More generally, the name space available for classes might not be sufficient.
5. *Objects and their classes might come into existence in reverse order.* As is argued in [8], a database user in a design environment like CAD creates objects in the first place, not type definitions or even classes. The usage of databases thus differs considerably from traditional applications where schema design has to be completed prior to instance creation.

We mention that one issue or the other from this list is sometimes reflected already in existing models, but never as a basic design target. We next sketch alternative approaches in which aspects like the above have been made a central objective.

Changing and Multiple Types

We consider proposals which address the issues of changing and/or multiple types first.

The perception that multiple, changing roles are typical of long-lived entities, and that primitives to model them are crucial to next-generation information systems, is one of the design goals of the *Melampus* system, under development at IBM Almaden

[7]. The data model of this system provides a mechanism named *aspects* [22] to allow objects to change their type and to model objects that have many types. In brief, an aspect extends an existing object with additional state and behavior while maintaining its identity; an object may enjoy many aspects simultaneously, and these can vary over time. By having many aspects, an object is an instance of many types, instead of being an instance of a unique class defined via multiple inheritance. Finally, object references are to particular aspects, and object behavior depends on the aspects referenced.

In the data model of Iris [30], objects are also allowed to acquire or lose types dynamically; however, an object retains its identity across type changes. In Iris, attributes of objects, relationships among objects, and computations on objects are uniformly expressed in terms of functions. When an object representing a person changes its type from, say, *Employee* to *Retiree*, all functions defined on the latter become applicable to it, while those defined on the former become inapplicable.

The next proposal, described in [25], uses concepts from *prototype* object-oriented languages (see below) and introduces *object hierarchies* as models for real-world entities; each object in a hierarchy represents some information about an entity. An object may have some state and some behavior, and additionally inherits behavior from its parent. When a message is sent to an object, it either responds directly, or *delegates* the message to its parent, so that inheritance is determined on a per-object basis. An existing hierarchy can be *cloned*, and an object can be extended with the behavior of another object. Finally, *template hierarchies* provide *prototypes* from which instances may be cloned.

Object Evolution

Prototype languages [20,28] generally suggest to model applications *without* a classification that partitions the world into entity sets. A prototype represents default behavior for some concept, and new objects can re-use part of the knowledge stored in a prototype by saying how they differ from it. Upon receiving a message an object does not understand, it can forward (delegate) it to its prototype to invoke more general behavior. In the area of object-oriented programming languages, many people believe that this approach has advantages over the class-based one with inheritance, with respect to the representation of default knowledge and incrementally and dynamically modifying concepts. The investigation of *classless* models in the context of object-oriented databases has only recently been proposed [27], but it appears to be a more appropriate way of modeling evolving objects like those arising in design environments.

A different proposal to capture the issue of object evolution has recently been made in [15], where the concept of *or-objects* is introduced. These are intended to model incomplete specifications, in which choices to be made during a design process are not fixed from the beginning. Or-objects are instances of *or-types*, which encapsulate

sulate different design alternatives, and thus allow storing incomplete information. Again, it seems that a model like this is more appropriate in a database for CAD or CASE than one of the form we described in Section 2. But while or-objects and their types have a formalization that extends the model of [1], formal models for classless approaches have yet to be proposed.

5 Conclusions

In this paper, we have given a personal account of current work on formal models for object-oriented databases. Although there is not a single uniform such model, the foundations on which such models have to be built seem understood. On the other hand, a number of interesting research issues still deserve further investigation.

In the process, we have also touched upon design issues, and in particular questioned whether the distinction between a conceptual and a logical design phase is still apt; on the other hand, an obvious open problem here is an integration of structural and behavioral design.

Finally, we have argued that formal models as they are currently available seem hardly suited for the nonstandard applications which initiated the consideration of object-orientation in the context of databases. A reason seems to be that many researchers have too much of a relational background, and try to exploit that as long as possible. As was done a number of years ago, when database people discovered what programming-language people had been studying since the days of Simula, it seems again necessary to take recent developments in the language area into account, and to adopt them for solving the problems database applications have.

References

- [1] S. Abiteboul, P.C. Kanellakis: The Two Facets of Object-Oriented Data Models; IEEE Data Engineering Bulletin 14 (2) 1991, 3-7
- [2] S. Abiteboul, P.C. Kanellakis, E. Waller: Method Schemas; Proc. 9th ACM Symposium on Principles of Database Systems 1990, 16-27
- [3] P.M.G. Apers et al.: Inheritance in an Object-Oriented Data Model; Memoranda Informatica 90-77, University of Twente 1990
- [4] H. Balsters et al.: Sets and Constraints in an Object-Oriented Data Model; Memoranda Informatica 90-75, University of Twente 1990
- [5] E. Bertino et al.: An Object-Oriented Data Model for Distributed Office Applications; Proc. ACM Conference on Office Information Systems 1990, 216-226
- [6] E. Bertino, L. Martino: Object-oriented Database Management Systems: Concepts and Issues; IEEE Computer 24 (4) 1991, 33-47

- [7] F. Cabrera et al.: The Melampus Project: Toward an Omniscient Computing System; IBM Research Report RJ7515, San Jose 1990
- [8] W. Cellary, G. Vossen, G. Jomier: Multiversion Object Constellations for CAD Databases; Techn. Report 9105, Division of Computer Science, University of Gießen, November 1991
- [9] K. Denninghoff, V. Vianu: The Power of Methods with Parallel Semantics; UCSD Technical Report No. CS91-184, University of California, San Diego, February 1991; extended abstract in Proc. 17th Int. Conference on Very Large Data Bases 1991, 221-232
- [10] O. Deux et al.: The Story of O_2 ; IEEE Transactions on Knowledge and Data Engineering 2, 1990, 91-108
- [11] M. Gyssens, J. Paredaens, D. van Gucht: A Graph-Oriented Object Database Model; Proc. 9th ACM Symposium on Principles of Database Systems 1990, 417-424
- [12] M. Gyssens et al.: A Graph-Oriented Object Database Model; Techn. Report, University of Antwerp 1991
- [13] R. Hull, R. King: Semantic Database Modeling: Survey, Applications, and Research Issues; ACM Computing Surveys 19, 1987, 201-260
- [14] R. Hull, K. Tanaka, M. Yoshikawa: Behavior Analysis of Object-Oriented Databases: Method Structure, Execution Trees, and Reachability; Proc. 3rd FODO Conference, Springer LNCS 367, 1989, 372-388
- [15] T. Imielinski et al.: Incomplete Objects — A Data Model for Design and Planning Applications; Proc. ACM SIGMOD International Conference on Management of Data 1991, 288-297
- [16] A. Kemper et al.: GOM: A Strongly Typed Persistent Object Model with Polymorphism; Proc. German GI Conference on "Datenbanken für Büro, Technik und Wissenschaft" (BTW) 1991, Springer Informatik-Fachbericht 270, 198-217
- [17] W. Kim: *Introduction to Object-Oriented Databases*; MIT Press 1990
- [18] C. Lecluse et al.: O_2 , an Object-Oriented Data Model; Proc. ACM SIGMOD International Conference on Management of Data 1988, 424-433
- [19] C. Lecluse, P. Richard: Foundations of the O_2 Database System; IEEE Data Engineering Bulletin 14 (2) 1991, 28-32
- [20] H. Liebermann: Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems; Proc. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) 1986, 214-223

- [21] U. Masermann: Design and Implementation of a Graphical Tool for the Design of Object-Oriented Database Schemata; Master's Thesis (in German), University of Koblenz-Landau, 1991
- [22] J. Richardson, P. Schwarz: Aspects: Extending Objects to Support Multiple, Independent Roles; Proc. ACM SIGMOD International Conference on Management of Data 1991, 298–307
- [23] M.H. Scholl, H.J. Schek: A Relational Object Model; Proc. 3rd International Conference on Database Theory 1990, Springer LNCS 470, 89–105
- [24] H.J. Schek, M.H. Scholl: Evolution of Data Models; Proc. Database Systems of the 90s, November 1990, Springer LNCS 466, 135–153
- [25] E. Sciore: Object Specialization; ACM Transactions on Information Systems 7, 1989, 103–122
- [26] D.D. Straube, M.T. Özsu: Queries and Query Processing in Object-Oriented Database Systems; ACM Transactions on Information Systems 8, 1990, 387–430
- [27] J.D. Ullman: A Comparison of Deductive and Object-Oriented Database Systems; Proc. 2nd DOOD Conference, Springer LNCS 566, 1991, 263–277
- [28] D. Ungar, R. Smith: Self: The Power of Simplicity; Proc. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) 1987, 214–242
- [29] G. Vossen, K.U. Witt: Objectbase Schemata and Objectbases in the FOOD Model; Techn. Report 9101, Division of Computer Science, University of Gießen, June 1991
- [30] K. Wilkinson et al.: The Iris Architecture and Implementation; IEEE Transactions on Knowledge and Data Engineering 2, 1990, 63–75