Formalization of OODB Models

Gottfried Vossen
Institut für Wirtschaftsinformatik, Universität Münster
Grevenerstraße 91, 48159 Münster

1 Introduction

Object-oriented data models represent a current endpoint in the evolution of data models [23]. Their formalization has been attempted in a variety of papers, including [5; 6; 19]. This short paper indicates what we consider the common intersection of these (and other) approaches; we list the relevant features and components, and give an idea of how to formalize the notion of an object-oriented database schema.

An object-oriented data model has to capture a variety of requirements [8; 27], which differ considerably from those that traditional data models have to meet. 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 relevant features 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.
2. It should provide a precise semantics for a data definition language.
3. It has to be composed of both a specification and an operational part.
4. It represents a computational paradigm as a basis for formal investigations.

In this short note, we do not present a comprehensive survey of formal models for object-oriented databases which have been proposed in the literature, but instead try to point out the fundamentals of how such models are obtained. The result can be considered as a framework in which the essentials of the object-oriented paradigm can be expressed concisely and further studied. Indeed, we give hints to various such investigations that have recently been undertaken.

2 Core Aspects of Formal OO Models

In this section, we describe what we perceive as the core aspects of various proposals for object models, and we do so by distinguishing structural from behavioral aspects. Thus, we generally consider *schemas*, the central notion of any conceptual database description, to be pairs of the form $S = (S_{\text{struc}}, S_{\text{behav}})$; in what follows, we first consider each component in isolation and then indicate how the two 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 achieved for structure, but not for behavior.

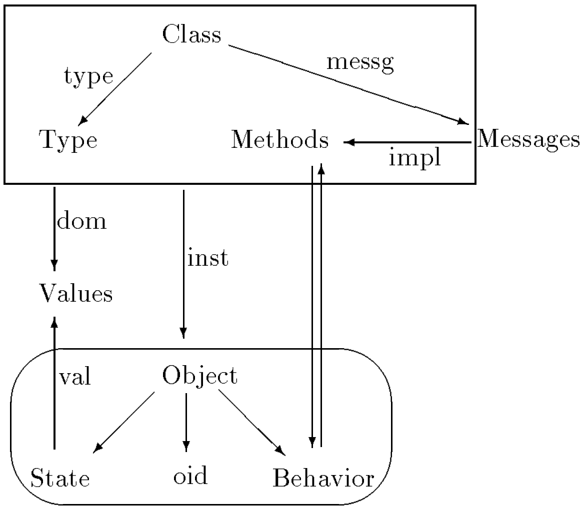

Figure 1: Core Aspects of an Object Model.

The core aspects of formal models for object-oriented databases are summarized in Figure 1, in which labels of arrows represent function names. In brief, the only structuring mechanism is the *class* which describes both structure and behavior for its instances, the *objects*. Structure is captured as a *type* for a class (in our notation, a function "type" associates a type with each class; the other function names shown above are to be interpreted similarly, see below). A type is nothing but a description of a *domain*, i.e., a set of values, and may or may not be named (in the former case, type names distinct from class names and attribute names must be provided). Values comprise the state of an object and can be as complex as the type system allows (i.e., depending on the availability of base types and constructors like tuple, set, bag, list, etc.). Behavior is manifested in a set of *messages* associated with each class (its external interface), which are internally implemented using *methods* that are executable on objects. Hence, objects have a state *and* a behavior; in addition, they are uniquely identified. Messages are specified by providing a *signature*, and by associating several signatures with the same message name, the latter gets *overloaded*. Not shown in Figure 1 is the possibility to organize classes in an *inheritance hierarchy*; also not shown is the fact that class attributes are allowed to reference other classes, thereby forming an *aggregation lattice*.

We next look at structural as well as behavioral aspects in more detail. 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
$[A_1 : t_1, \ldots, A_n : t_n] \in T$ ("tuple type");

(iii) if $t \in T$, then $\{t\} \in T$ ("set type");

(iv) if $t \in T$, then $< t > \in T$ ("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. Notice also that here types are not named; for practical reasons, the use of type names may be desirable (e.g., in order to be able to reuse type definitions in various places throughout a schema), and if it is, it can easily be added to the above in the way indicated earlier.

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

(a) dom(`integer`) is the set of all integers; dom is analogously defined for `string, float, boolean`;

(b) $\text{dom}([A_1 : t_1, \ldots, A_n : t_n]) :=$
$\{[A_1 : v_1, \ldots, A_n : v_n] \mid (\forall\, i, 1 \leq i \leq n)\ v_i \in \text{dom}(t_i)\}$;

(c) $\text{dom}(\{t\}) :=$
$\{\{v_1, \ldots, v_n\} \mid (\forall\, i, 1 \leq i \leq n)\ v_i \in \text{dom}(t)\}$;

(d) $\text{dom}(< t >) :=$
$\{< v_1, \ldots, v_n > \mid (\forall\, i, 1 \leq i \leq n)\ v_i \in \text{dom}(t)\}$.

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) $\text{dom}(c) = OID$ 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 `integer` type takes integer numbers as values). Clearly, this alone is not enough, since class instances commonly have distinct sets of object identifiers associated with them. We will show below how that (and, for example, the fact that sometimes inclusion dependencies need to hold between sets of class instances) is captured at the instance level.

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; for example, it can be defined semantically by requiring that the sets of values or instances of types, where one is a subtype of the other, are in a subset relationship. We prefer a simpler, syntactical approach, which has, for example, the advantage that checking subtype relationships can be automated:

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) $[A_1 : t_1, \ldots, A_n : t_n] \leq [A'_1 : t'_1, \ldots, A'_m : t'_m]$ 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) $< t > \leq < t' >$ 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) type : $C \to T$ is a total function associating a type with each class name,

(iv) isa $\subseteq C \times C$ is a partial order on C which is consistent w.r.t. subtyping, i.e.,
c 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 [17; 19; 20; 25] 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, like attributes as functions [22; 29], a distinction of class attributes from instance attributes (the latter are 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) [7], a unique root of the class hierarchy from which every class inherits [20], a distinction between private and public attributes [12], 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 [18]), 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). For another example, the ODMG-93 proposal for a standardized model [10] contains explicit keys, (binary) relationships, and inverse attributes. None of these features appear in our model, the reason being that these are *not* specific to object-orientation.

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 $\text{sign}(m) = \{s_1, \ldots s_l\}$, $l \geq 1$, of signatures; each s_h, $1 \leq h \leq l$, has the form $s_h : c \times t_1 \times \ldots \times t_p \to t$ for $c \in C$, $t_1, \ldots, t_p, t \in T$
(each signature has the receiver of the message as its first component),

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

(iv) messg : $C \to 2^M$ is a mapping s.t. for each $c \in C$ and for each $m \in \text{messg}(c)$ there exists a signature $s \in \text{sign}(m)$ satisfying $s[1] = c$,

(v) impl: $\{(m, c) \mid m \in \text{messg}(c)\} \to P$ is a partial function.

In combining structural and behavioral schemas, we finally obtain an *objectbase schema* of 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 isa c' implies $\text{messg}(c') \subseteq \text{messg}(c)$ for all $c, c' \in C$,

(ii) if c isa c' and $s, s' \in \text{sign}(m)$ for $m \in M$ such that $s : c \times t_1 \times \ldots \times t_n \to t$, $s' : c' \times t'_1 \times \ldots \times t'_n \to t'$, then $t_i \leq t'_i$ for each i, $1 \leq i \leq n$, and $t \leq t'$,

(iii) for each $m \in \text{messg}(c)$ there exists a $c' \in C$) s.t. c isa c' and $\text{impl}(m, c')$ 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, and is called the *covariance condition* in [20; 9]. The covariance condition is a significant difference from what is used at a corresponding point in programming languages, and which is known as the *contravariance* condition; for a detailed explanation, see [9]. 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 [20]. More formally, if $m \in M$ and $s : c \times t_1 \times \ldots \times t_n \to t \in \text{sign}(m)$, then $\text{impl}(m, c)$, if defined, is a program $p \in P$ of the form

$$p : \text{dom}(c) \times \text{dom}(t_1) \times \ldots \times \text{dom}(t_n) \to \text{dom}(t)$$

The condition in question informally states that if message overloading appears in isa-related classes (so that the corresponding signatures satisfy the covariance 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 $s : c \times t_1 \times \ldots \times t_n \to t$, $s' : c' \times t'_1 \times \ldots \times t'_n \to t'$, c 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 $\text{dom}(c) \times \text{dom}(t_1) \times \ldots \times \text{dom}(t_n)$.

A variety of formal investigations for behavioral schemata in the sense defined above can already be found in the literature, which investigate questions including 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, or potential parallelism of method evaluations. To investigate such issues, our general notion of schema is made precise in various ways. For example, [15] 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

[11] 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 [11] looks at tractability guarantees corresponding to those known for relational query languages. Also, it is pretty straightforward to define an *object algebra* for a model like the one sketched in the previous section; see, for example, the papers in [13]. That carries over to issues like query optimization, implementation of operations, and query processing. A survey of other recent investigations that have similar bases or origins can be found in [28].

We emphasize again that the model just sketched can be seen as description of the core of vastly any object-oriented model; however, this is valid only relative to the fact that many specialities, which have been proposed in the literature, or which are being built into commercial systems, are neglected here.

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) $O \subseteq OID$ is a finite set of object identifiers,

(ii) 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 isa c', then $\text{inst}(c) \subseteq \text{inst}(c')$,

(iii) val: $O \rightarrow V$ is a function s.t. $(\forall\, c \in C)\,(\forall\, o \in \text{inst}(c))\, \text{val}(o) \in \text{dom}(\text{type}(c))$.

Notice that this definition closes the problem left open earlier, namely that class domains originally were simply the set *OID*.

3 Open Issues

We next survey several modeling issues in object-oriented databases which have not yet received enough research 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 [21].
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* [26], 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.* 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. Alternative approaches, which takes these issues into consideration right from the start, appear, for example, in [21; 24; 16]. A possible general concept for the solution of these problems seems the exploitation of prototype languages, which 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 in [26], and a concrete model is reported in [14].

4 Conclusions

In this short paper we have tried to give a rough personal account of recent 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, and even standardization efforts have recently been launched [10]. On the other hand, a number of interesting research issues still deserve further investigation. In particular, 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; this is more than confirmed by the ODMG-93 proposal. As was done a number of years ago, when database people discovered what programming-language or knowledge-representation people had been studying for years already, it seems again necessary to take recent developments in these areas 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] F. Bancilhon, C. Delobel, P. Kanellakis (eds.): *Building an Object-Oriented Database System — The Story of O_2*. Morgan-Kaufmann 1992

[6] C. Beeri: A Formal Approach to Object-Oriented Databases; Data & Knowledge Engineering 5, 1990, 353–382

[7] E. Bertino et al.: An Object-Oriented Data Model for Distributed Office Applications; Proc. ACM Conference on Office Information Systems 1990, 216–226

[8] E. Bertino, L. Martino: Object-oriented Database Management Systems: Concepts and Issues; IEEE Computer 24 (4) 1991, 33–47

[9] E. Bertino, L. Martino: *Object-Oriented Database Systems*; Addison-Wesley 1993

[10] R.G.G. Cattell (ed.): *The Object Database Standard: ODMG-93*. Morgan-Kaufmann 1994

[11] 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

[12] O. Deux et al.: The Story of O_2; IEEE Transactions on Knowledge and Data Engineering 2, 1990, 91–108

[13] J.C. Freytag, D. Maier, G. Vossen: *Query Processing for Advanced Database Systems*; Morgan-Kaufmann 1994

[14] M. Groß-Hardt, G. Vossen: *Towards Classless Object Models for Engineering Design Applications*; Proc. 4th International Conference on Database and Expert Systems Applications (DEXA) 1993, Prag, Springer LNCS 720, 36–47

[15] 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

[16] 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

[17] 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

[18] W. Kim: *Introduction to Object-Oriented Databases*; MIT Press 1990

[19] C. Lecluse et al.: O_2, an Object-Oriented Data Model; Proc. ACM SIGMOD International Conference on Management of Data 1988, 424–433

[20] C. Lecluse, P. Richard: Foundations of the O_2 Database System; IEEE Data Engineering Bulletin 14 (2) 1991, 28–32

[21] J. Richardson, P. Schwarz: Aspects: Extending Objects to Support Multiple, Independent Roles; Proc. ACM SIGMOD International Conference on Management of Data 1991, 298–307

[22] M.H. Scholl, H.J. Schek: A Relational Object Model; Proc. 3rd International Conference on Database Theory 1990, Springer LNCS 470, 89–105

[23] H.J. Schek, M.H. Scholl: Evolution of Data Models; Proc. Database Systems of the 90s, November 1990, Springer LNCS 466, 135–153

[24] E. Sciore: Object Specialization; ACM Transactions on Information Systems 7, 1989, 103–122

[25] D.D. Straube, M.T. Özsu: Queries and Query Processing in Object-Oriented Database Systems; ACM Transactions on Information Systems 8, 1990, 387–430

[26] J.D. Ullman: A Comparison of Deductive and Object-Oriented Database Systems; Proc. 2nd DOOD Conference, Springer LNCS 566, 1991, 263–277

[27] G. Vossen: *Datenmodelle, Datenbanksprachen und Datenbankmanagement-Systeme*; 2. Auflage, Addison-Wesley 1994

[28] G. Vossen: Database Theory: An Introduction; Technical Report, University of Münster, June 1994

[29] K. Wilkinson et al.: The Iris Architecture and Implementation; IEEE Transactions on Knowledge and Data Engineering 2, 1990, 63–75