Why Isn't There an Object-Oriented Data Model?

David Maier

Oregon Graduate Institute
Department of Computer Science and Engineering
20000 NW Walker Rd
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E-89-002
April 23, 1989

Why Isn't There an Object-Oriented Data Model?

David Maier

Oregon Graduate Center
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 89-002

Report published in *Proceedings of the IFIP 11th World Computer Conference*, San Francisco, California, August-September 1989.

Why Isn't There an Object-Oriented Data Model?

David Maier

Abstract

The appellation "object-oriented" is affixed to a wide range of recent database system prototypes and products, with no agreement on its exact meaning. This paper offers a definition for an object-oriented *database system,* explains why a standard for "the" object-oriented *data model* will be hard to come by, and examines the differences in the models of current object-oriented databases. It concludes with predictions on the effect of object-oriented databases on the commercial marketplace.

WHY ISN'T THERE AN OBJECT-ORIENTED DATA MODEL?

David MAIER

Dept. of Computer Science & Engineering, Oregon Graduate Center
19600 NW von Neumann Drive, Beaverton, Oregon 97006, USA
e-mail: maier@cse.ogc.edu

1. INTRODUCTION

Object-oriented database systems (OODBS's) have come from the idea stage to second-generation prototypes and commercial products in the past five years. Several international workshops have been devoted to OODBS's [OODB86, OODB88] and related topics [Appin, DBPL, POS87, POS89]. Yet, in spite of all the activity, no standard data model for OODBS's has emerged. Why isn't there an Object-Oriented Data Model, in the spirit of the relational, hierarchical or network data models, or the more semantic entity-relationship model?

I wish I could answer the question simply by proposing such a data model, but I believe that there are fundamental reasons why such a model cannot be formulated, at least along the lines of current data and semantic models. In this paper I discuss those reasons, which include extensible type systems, the great variety of modeling features in OODBS's and even the problem of determining what "data model" means in the context of an OODBS. I do believe there can be consensus on the requirements that a database *system* wanting to call itself "object-oriented" should satisfy, and I summarize such a list that has been proposed elsewhere. At the end of the paper I speculate on the future for OODBS's in the commercial marketplace.

2. WHAT IS AN OBJECT-ORIENTED DATABASE?

Before defining an object-oriented database system, a few words on why OODBS's are appearing. One reason is that there are application areas not well served by conventional systems. Applications such as computer-aided design, knowledge bases and office information systems have representation demands that do not mesh well with the capabilities of conventional database systems. Such applications are characterized by the complexity of the data, the need for new data types, such as bit-maps and large blocks of text, and the need to capture complex semantics of interpreting and updating the data. OODBS's attempt to meet these needs. A second problem I have termed the *impedance mismatch* problem [BM]. Most database applications involve two languages, the data manipulation language of the database system, and a general purpose programming language. These languages are almost always mismatched in their type systems and their programming style. OODBS's

provide more powerful data languages, so that more, or even all, of an application can be coded in one language, avoiding crossings of the boundary between two dissimilar languages.

There is no wide agreement on what constitutes an object-oriented database system. As my preamble suggests, defining an object-oriented database system as one that implements the object-oriented data model begs the question. For this paper, I will adopt a definition of an OODBS put forth in a recent position paper [ABD+]. In capsule, an OODBS supports *objects*, which are chunks of private state with a public interface of operations. The objects are grouped into *types* or *classes* on the basis of shared interfaces or implementation. (I will mostly avoid the term *class* in the remainder of the paper, as it has so many meanings in the object-oriented literature.)

In discussing these features, I will use a running example. I want to represent information about roads, and relate it to coordinates on a particular street map. The main entities will be *roads*, *road segments* and *intersections*. Intersections divide a road into a sequence of road segments. Each road segment has a beginning and ending intersection. Each intersection represents the convergence of two or more road segments. Each road is described as an alternating sequence of road segments and intersections. Each road segment knows which road it belongs to. (There are other ways to capture the same information; I make no claim my scheme is optimal.) Intersections have an associated position given in map coordinates. Road segments have a shape, for display purposes, which may either be a straight line or a curved arc. Each also has an address range, a side for odd-numbered addresses and a direction (two-way or one-way). Each road has a name and a road type (street, avenue, ...). This information is summarized in the description below, in no particular schema language.

```
Road
   name: String
   roadType:  road, avenue, boulevard, lane ...
   segmentList: sequence of RoadSegment or Intersection

RoadSegment
   inRoad: Road
   segmentShape: Arc or Line
   start: Intersection
   end: Intersection
   direction:  one-way-SE, one-way-ES, two-way
   addressRange: (StreetNumber, StreetNumber)
   oddSide:  left, right

Intersection
   incomingRoads: set of RoadSegment
```

position: Coordinate

Here **Coordinate**, **StreetNumber**, **Arc** and **Line** are types assumed defined elsewhere.

The list of requirements for an OODBS follows.

1. **Support for Complex Objects.** In representing the internal state of an object, we can freely compose data structures. The internal state of an object can have fields (often called *instance variables*) that hold records, sets, arrays or other objects. In our example, the **position** field of an **Intersection** object is a **Coordinate** object, and the **segmentList** field of a **Road** object is a heterogeneous sequence of **RoadSegment** and **Intersection** objects.
2. **Object identity.** Every object has a system-supplied identity that is distinct from its internal state. The identity is maintained over changes in the internal state, and two distinct objects can have the same internal state. Our example would allow two **RoadSegment** objects between the same pair of **Intersection** objects. An object identity must be sufficient by itself to identify an object within the database system. Hence, a key in a relation does not qualify as an object identity, because it must be used in conjunction with the relation name to identify a tuple.
3. **Encapsulation of Behavior.** The database designer can define operations on objects. The code for the operation, called a *method*, and the internal state of the object are packaged together. The state of the object is no longer directly manipulable, but only be accessed or modified by invoking one of its operations. (Such an invocation is sometimes called a *message*.) The set of operations that an object accepts is called the *protocol* for the object. Thus, the semantics of access and update are captured in the operations. Encapsulation is similar to abstract data types, except that ADTs are usually discussed in a functional language setting, whereas methods may modify object state. In our example, we might encapsulate the structure of a **Road** object with operations to add and remove a **RoadSegment** or **Intersection**, to return a count of **RoadSegments**, to return the name of the road, and so on.
4. **Types.** Rather than each object carrying around its own protocol, objects with the same behavior can be grouped together and share this information. This grouping is called a *type* or *class*, usually. (The term *class* usually implies implementation information is associated with the grouping, whereas *type* can mean only the specification of the interface to an object, with multiple implementations possible.) Objects are then instances of a type, and each derives its behavior from its type. There can be variation on whether a type is a purely intensional construct—just specifying behavior of instances—or also extensional—serving to keep a collection of

all extant instances. The former variant is like a programming language type, the latter resembles more a relation declaration.

5. **Hierarchies.** The main point here is to take advantage of similarities among categories of objects, for modeling, implementation, integrity or querying. I describe below three varieties of hierarchies an OODBS might support. In each description, assume category **Ai** is above category **Bi** in the hierarchy.

   Specification Hierarchy: If **A1** and **B1** are in a specification hierarchy, it means that an instance of **B1** can be used where an instance of **A1** is expected. Thus, **B1** supports at least the protocol of **A1**. An instance of **A1** might be expected are as the value of a field in a record, in a temporary variable, or as an argument to a method. When **A1** and **B1** stand in this relationship, it is common to say that **B1** is a *subtype* of **A1**. Consider a category **ControlledIntersection**, below **Intersection** in a specification hierarchy, where an instance of **ControlledIntersection** represents an intersection with a traffic signal. A **ControlledIntersection** object might support additional messages to get information about the signal. With this example, the **start** field of a **RoadSegment** object could actually hold a **ControlledIntersection** instance, even though **RoadSegment** is declared as an **Intersection,** since such an object will understand all the messages that might be sent to a **Intersection** object.

   Implementation Hierarchy: Here we have categories **A2** and **B2** where objects from **B2** use the methods and internal representation of **A2**, possibly with additions or modifications. Such reuse of implementation is often termed *inheritance* of behavior. For example, we might want a category **Route** whose objects have much the same structure and methods as **Road** objects, but where **Route** is not a subtype of **Road**. (Perhaps **Route** masks the **roadType** information or gives different names to the methods of **Road**.)

   Extent Hierarchy: Here **A3** and **B3** are interpreted as collections of instances, and the relationship is that **B3** is a subset of **A3**. For example, we could have a collection **PortlandSegments** of all road segments in the Portland city limits, and a subset **PtlndRepairSegs** that contains Portland road segments that are currently under repair.

   This list is not exhaustive. Other hierarchies exist, such as subpart hierarchies and constraint hierarchies, although explicit support is not as common. An OODBS should provide at least specification and implementation hierarchies, and support extents (although not necessarily in a hierarchy). One problem is that many object-oriented systems use one mechanism to support two or more of these hierarchies. In particular, combining the specification and implementation hierarchies is common,

and often called a *class* hierarchy.

6. **Late Binding.** The declared type of a variable in a method might differ from the "immediate type" of the the variable: the type of the object actually occupying the variable at runtime. This situation can occur with a specification hierarchy, when the immediate type is a subtype of the declared type. The implementation of a given operation can be different for the declared type and the immediate type. For example, the **plot** message to an **Intersection** object might return a bit-map icon depicting the intersection. **ControlledIntersection** can reimplement this message. At the time a piece of code is compiled, it may be impossible to determine what the immediate type of a particular variable will be at runtime. (Indeed, the immediate type of a variable may change from one moment to the next. Consider a variable that is being set to successive **Intersection**s in a **Road** in turn. Some of the intersections will be controlled intersections, while others will not.

7. **Computational Completeness.** The language for writing methods is Turing-complete, so that any computable sort of behavior can be captured in a method.

8. **Extensibility of the Type System.** The set of types that the system supports can be extended (this is implicit in earlier points). Moreover, user-defined types are manipulated with the same syntax as system-supplied types, in the data language.

9. The last requirement of an OODBS is that it is truly a database system. It must provide

   Persistence: Any object can live past the life of the process that creates it. In most programming language, only files are persistent.

   Secondary Storage Management: The database system handles the movement of data between main memory and disk.

   Concurrency: Multiple users can access the database simultaneously.

   Recovery: Protection is provided against process, processor and media failures.

   Ad Hoc Query Support: There is a facility for quickly formulating queries against the database. Such a facility need not look like a relational query language. It should be (i) high-level (emphasizing what is wanted over how to derive it), (ii) reasonably efficient (some effort is made to avoid the worst strategies for evaluating a query), and (iii) generic (queries can be posed against instances of a type as soon as the type is added to the database). Thus, some graphical browsing tools will fit the bill for ad hoc query support.

A number of systems fit this list of requirements, or come close to it. Examples of commercial systems are GemStone [MSOP], Vbase [AH] and VISION [CS].

Some research prototypes are Alltalk [RMS], Cactis [HK], Encore/Observer [WZ], Extra/Excess [CDV], IRIS [Fi+], ODM [KB], ORION [BC+], O2 [BBB+], Probe [MD], PS-Algol [ABC+] and TEDM [ZM]. (See also [DBE].)

Many of these systems have a two-layer architecture, which is also common in relational systems. The bottom layer is a *storage manager.* It usually has a purely structural type system, and handles concurrency, recovery, associative access, authorization and resource allocation. On top of the storage manager is the *execution layer,* which implements a more sophisticated type system, enforces encapsulation and executes methods, making storage manager calls as needed. There is typically one instance of the storage manager in the database, but an instance of the execution layer for each active session. In early systems, both layers had to run on the same processor. More recently, some OODBS's are able to distribute the layers, with the storage manager running on a central server, and the execution layers running on individual workstations. If the execution layer does sufficient buffering, such local distribution can be an enormous performance boost [RKC].

3. WHAT IS A DATA MODEL?

The term *data model* covers two broad concepts in the database world. It can mean a particular schema or description for the information used in a data-intensive application. (In this sense the term is also used as a verb: "data modeling" means designing the schema for a particular application.) The other meaning is the system or language in which schemata or descriptions are expressed. I intend the latter meaning of "data model" here.

Even this meaning is has multiple senses. It can be construed concretely as the data definition facility of a single database system, or more broadly, as the common core shared by a group of similar database systems: the data definition language of ORACLE in particular versus the relational model in general. It can be interpreted shallowly as a particular data definition language, or more deeply as a mathematical algebra or logic underlying such a language. I believe it is the broad, deep connotation that people intend when asking for an object-oriented data model. Certainly, if we look at a particular OODBS, such as Vbase, we see a particular data description language (in that case TDL). However, if we look across different OODBS's, no common mathematical structure emerges that underlies their data definition or manipulation languages, unlike relational database systems.

What constitutes a data model, in the sense of the relational data model or the network data model? A data model is a collection of data structures, operations over those structures, and constraints on states of those structures [LT, MER]. Typically a data model includes some scalar types, record structures, and set structures (relational), tree structures (hierarchical) or headed-list structures (network). The data structures are parameterized: we can define records over different fields, or tree structures with different records, or lists with

different header and member types. Perhaps a more accurate view of the data structures is as type constructors, except they do not compose freely. For example, a record of records is usually forbidden.

Is it accurate to characterize the data model as just the type system for the data manipulation language? A data model is not exactly like the type system of a programming language. One confusing property of traditional data models is that the data structures determine the data types. That is, once we have defined the data structures for a particular database scheme, the operations on those structures come automatically, and cannot be extended within the data language. For example, in defining a record structure, we get an operation for creating a new record with that structure, and operations to set and get fields in that type of record. With the trees in the hierarchical model come operations such as "get first" and "get next within parent". There is seldom any means in the data language to define a new operation on the structure, or if the means exists, it is limited—a virtual field, or a logical link. The operations are derived from the data structure, so the data type (structure plus associated operations) is fixed by the structure. Bloom and Zdonik [BZ] examine this and other cultural differences between the database world and the programming language world.

Existing data models are not true type systems. They are definitely not abstract data type systems. A more accurate description is that a data model is a fixed set of parameterized data types. During database schema definition, these parameterized data types are specialized, by giving the fields in a record, the levels in a hierarchical tree, and so forth. Then the database is populated with instances of these concrete types. An interesting point is that sometimes only one instance of a type is created. In the hierarchical model, there is one tree per tree type. In the relational model, there is one instance of each relation type (but many instances of the tuple types). In the data definition languages for most systems, one often defines a type and declares a unique variable of the type in the same statement. **Relation R(A, B, C)** defines a tuple type, a relation type and declares **R** to be a relation of the relation type. In programming language type systems, type definition and variable declaration are independent and orthogonal. We can declare any number of variables of any type. Contrast this to the situation in current database systems, where there is only one variable of a type, and some types cannot support (persistent) variables. I know of no relational database system that supports variables declared of tuple types (except range variables in queries, which are temporary).

4. WHY IT IS HARD TO DEFINE THE DATA MODEL OF AN OODBS

A data model in the current sense is a fixed collection of data types, possibly parameterized, plus some language for constraints. OODBS's give the ability to define new data types. The types are not dictated by the data structures.

Rather, data structures are used to represent instances of the type, but the operations for the type are defined explicitly, not defined implicitly from the data structures. What might we choose to mean by the data model of an OODB?

One possibility is the collection of constructors for forming the data structures in the representations: reference, set, array, record or whatever. This choice is not very informative. It does not show how one thinks of the data in the system. Different collections of representation constructors can support the same set of abstract types at the application interface. This choice characterizes an OODBS by its implementation options for types, which is tantamount to characterizing a conventional system by access methods available.

Another possibility is that the data model of an OODBS is the collection of types supplied with the system. There are several problems with this definition. Data models under this definition are hard to compare, as it in not always apparent if two data types are the same. Since an OODBS should have an extensible set of types, equivalence of the initial type collections does not imply equivalence of the collection of types in use in a given application. Further, the initial collection of types is likely to change over time. Manufacturers will add new types as standard.

A third possibility is to interpret "data model" as the type definition system of an OODBS. It certainly makes sense to compare these, but as argued above, this interpretation of data model is at odds with the conventional interpretation. It is a way to compare OODBS's with each other, but it will not work well for comparing OODBS's with the traditional data models.

Using the last proposal for a definition, what would it take to describe the data model of an OODBS? A start would be the representation constructors, plus the encapsulation mechanism, plus the collection of system-supplied types. Can there be agreement on a standard data model (type system) for OODBS's? Probably not on one, but maybe on some main ones. There will be models based on type systems of particular languages, such as C++ or Common Lisp. But such a data model will not suit all applications. It makes the match to one programming language good, at the expense of making the match to other programming languages harder. What kind of OODBS type system would work well with multiple programming languages [O'B]? One approach is a vanilla type system: the intersection of the type system of several languages, or a collection of types that map easily into all the programming languages. It would be a lingua franca for program structures. The advantage to this approach is that we could pass around data items that have more semantics than files. This approach promotes sharing between applications in different languages, but restricts the programmer in any one language. The other extreme is the pistachio with whipped cream, nuts, chocolate sauce and a cherry type system: the union of type systems from multiple languages. Each programming language could store all its data values directly. The problem is that an object

created in one language could not be read easily into another language. My conclusion: there's just not enough experience to standardize a type system yet.

In thinking about OODBS data models as type systems, the question arises of whether the method language is part of the standard. Could we standardize a type system apart from a language? (A related question is whether there could a OODBS's that supports databases with methods written in multiple languages [BM].)

5. SOURCES OF VARIATION

The main problem of standardizing an OODBS type system is the sheer variety of features in current systems. I list some ways current OODBS's vary, particularly in their type systems. Many of the variations arise because of differences in intended use among the systems (for example, embedded in a tool versus running stand-alone on a server).

1. **Representation Constructors.** While the set of types provided by the database to an application programmer should be extensible, the set of constructors for defining the internal representation of objects could be fixed. Typical constructors are record, array, sequence, set and reference. The precise set of constructors varies from system to system. Nested relation systems merge the record and set constructors. Some have insertable lists to provide efficient edits to ordered collections. The EXODUS system [CD+], through its **own** declaration, essentially allows using the definition of one type as a macro in defining the states of other types.

2. **Protocol Description.** Differing amounts of specification can be provided in the protocol for a type. The protocol can be nothing more than the set of messages that instances of the type understand. It can include the type signatures of the operators, or go further and specify axioms relating different operators [SS]. A useful relationship to know for maintaining auxiliary access paths is which operations can cause the results of what other operations to change. For example, knowing that the **getLocation** message to an **Intersection** object returns the same result until a **setLocation** message is processed is useful in maintaining an index on **getLocation**.

3. **What is Typed.** Typing information can be associated with object instances, with variables, or with both. An object might have more than one type. In the Vbase system [AH], the language for specifying legal values for fields is more expressive than the type system for object instances. For example, a field value can be restricted to belong to one of two types, but there is no union type constructor.

4. **Hierarchies.** OODBS vary in what hierarchies are supported and the exact semantics of those hierarchies. Sometimes the hierarchies must be strict tree structures, while other systems allow directed acyclic graphs.

5. **Encapsulation.** While encapsulation is a requirement, OODBS vary in what is inside the encapsulation envelope. In Smalltalk-like systems, only one layer of substructure is directly accessible inside the encapsulation envelope. So, if one wants to construct an object state that is an array of records, only the array can be manipulated structurally. The records will be separate objects that must be accessed through their protocol. Lomet [Lo] gives a system where arbitrary substructure of an object can be in the envelope. Also, the envelope might include just the state of one instance, or the states of all instances of a type, as in a CLU cluster [Li].

6. **Versions.** A valuable feature for design support is versions of objects. Version histories can be linear or branching [EE+], and a variety of flavors of reference exist relative to versions of an object: a fixed version, the latest version, all versions.

7. **Name Spaces.** In some OODBS's, only collections or types can be given names that persist in the database, while other systems allow objects of arbitrary types to be given persistent names.

8. **Self-Reference.** Some systems have type describing objects (TDOs) that hold type definitions and that are visible at runtime, much like relational systems that make schema information available in system relations. In systems with TDOs, the TDOs can be read-only, or they can be updatable. Being able to update a type definition at runtime severely hampers the ability to type check methods at compile time.

9. **Parameterization and Polymorphism.** An OODBS might support parameterized types. The support can be for just a fixed set of system-supplied types, or it can include facilities for users to define their own parameterized types. Polymorphism is the ability of a single piece of code to operate on structurally dissimilar objects. The most common kind of polymorphism is "subtype polymorphism" [Ca, CW], in which code for a supertype works on the extended structure of the subtype. More general types of polymorphism are possible. Polymorphic analogs of relational algebra operators have been defined for object-oriented systems; they require a sophisticated type system in which to be typed [BO].

10. **Extensibility.** The type system than an application programmer sees must be extensible in an OODBS, but the suite of representation constructors, storage structures and access routines might also be extensible [BBG+, CD+, LMP].

6. WHAT INFLUENCE WILL OODBS'S HAVE IN THE COMMERCIAL MARKETPLACE?

Here I offer my informed guesses on how OODB ideas and technology will emerge in the commercial marketplace over the next 3-4 years.

6.1. Relational Extensions

Probably the first place where current users of mainframe database systems will see an object-oriented flavor is in extensions to current relational systems (as in the GEM extension to QUEL [Za]). The most likely extension will be simple support for complex objects [HL]. Attributes of relations will be declared as references to tuples in other relations, and path notation in the query language will make such connections easy to follow. For example, to represent the road and intersection information from our running example, we might use several relations with reference attributes:

```
Roads(name: String, roadType: String, ...)
RoadSegments(inRoad: ref Roads, start: ref Intersections,
             end: ref Intersections, ...
Intersections(xPosition: Integer, yPosition: Integer, ...)
```

An SQL-type query language might extend the dot notation to traverse these connections:

```
select r.inRoad.name
from Roads r
where r.end.xPosition   155
```

There are several ways to implement such a feature. The references could be foreign keys to the other relations, or they could be internal tuple IDs. The joins to traverse the references can be supported by indices, direct links or clustering. The advantages of this extension is that it will eliminate about 90% of the explicit joins used currently, it will simplify many queries, it goes hand in hand with referential integrity constraints, and having attributes declared as a reference is a good hint as to what auxiliary structures are worthwhile to maintain [JF+].

Another likely addition is some type of relation hierarchy, such as in Postgres [SR, St87]. For example, we might define highways as a specialization of roads with an additional attribute for highway number:

```
Highways   Roads(HighwayNo: String)
```

Such a declaration might pertain to types (a **Highways** tuple can be provided where a **Roads** tuple is expected) or it might pertain to extensions (all **Highways** tuples are visible in the **Roads** relation, suitably projected) or both. This extension avoids null values in trying to model similar, but not identical, entity types. It also captures certain useful constraints between relations.

A third likely extension is user-defined data types as attribute values [OH, St86, WS+]. For example, in the road example, shapes of road segments might be described by some user-defined geometric data type. Note that such extensibility is not the same as the type extensibility called for in the feature list for OODBS's. In the proposed schemes for adding user types to relational systems, the added types have the same status as system-supplied base types, such as integer, string and float. Instances of these added types are internally uninterpreted by the database system; they cannot have references to other database entities as part of their representation, nor are the operations of the added types defined with the database language.

Storing packets of behavior in the database is already starting to appear in commercial systems, although not encapsulated so as to form abstract data types. The main reason I see for this added feature is the performance boost obtained from being able to invoke a series of database commands with one call to the database. For example, the TP1 benchmark takes four or five separate DML commands to specify in most relational languages. Being able to store that sequence of commands in the database reduces five application-to-database calls to one.

Along the lines of these extensions, some kind of standard "Object SQL" seems feasible to me. However, a database supporting such a language would not necessarily be an OODBS by the definition offered earlier.

As relational systems start trying to serve the design database market better, I expect they will start mimicking the partitioned workstation-server architectures popular in OODBS implementations. Doing some query processing and buffering locally to the design tools is necessary to approach the speeds on single-object navigation gotten with reading the design structures into program memory [Ma]. There is already movement in this directions with personal computer front-ends to mainframe databases.

6.2. Semantic Models

The past year saw the introduction of a commercial product with a semantic data model: SIM [JF+], which is based on SDM [HM]. SIM incorporates many of the features mentioned in the relational extensions section above. Many semantic data models include some of the features of OODBS's already, and can be extended to include more of them. The success of commercial products based on semantic models depends wholly on the systems exploiting the added schema information for improved performance.

6.3. Persistent Versions of Existing Languages

There is a demand for a programming system for design environments with a very low impedance mismatch between programming language and database support. Thus, database systems will appear that extend an existing language so that data structures can persist. The first products will be some objectified

version of C, such as C++. Some CAD tool companies already have their own internal extensions for for object management, although they are usually concerned with just mapping data between main memory and disk, and do not provide all the features of a database system, such as concurrency control and recovery. There are four companies I know of working on database systems that will integrate closely with C++: Ontologic, Object Design, Object Sciences and Objectivity. Each of their efforts will provide in some way for persistent C++ objects, but will likely differ in added features for object versions, reference flavors, configurations (consistent sets of versions), environment capture (such as what version of what tool was used to create this VLSI cell), dependency tracking (which cell libraries does this chip use), concurrency control techniques, support for cooperative work and design partitioning and distribution.

The other place that extensions for persistence are appearing are for LISP systems, to support large knowledge bases for both expert systems and design applications [Fo+, WF+].

Initially, the users of such extended languages will be fairly sophisticated tool writers, although some sort of ad hoc query facilities will likely appear in later releases. I see such database systems creating a new market more than capturing much of the market away from more conventional database systems.

6.4. Structural Object-Oriented Systems

European development efforts on OODBS's have roots in semantic models, whereas North American efforts derive more from work in object-oriented programming languages. (Nascent Japanese efforts emphasize the connection to logic programming.) Thus, European systems are mostly structural models, without the encapsulation of behavior. There are both nested-relation systems and systems with more direct modeling of object identity [Be, Da+, DGL, PS+]. These systems are now moving into the beta-test stage and beyond, being used as components for academic-industry research projects, such as common programming environments. In the intended model of use, the architecture ends up being not so far from the encapsulated behavior OODBS's: A layer is written on top of the structural model that attaches behavior to the structures, for use within a particular domain of applications, such as programming environments, geographic information systems or graphics. This layer runs under the end application on individual workstations, with the structural database on a central server. So the end architecture looks much like that of behavioral OODBS's that have a storage manager component and a language evaluation component. The difference is how tightly coupled the layers are. The European approach has the advantage that it allowed the workstation-server distribution to happen sooner than with the behavioral OODBS's. It also admits the possibility of having different behaviors attached to the same structures for different applications, by going through a different semantic layer [Be]. The semantic

layer also can control batch movement of objects from server to workstation.

6.5. Existing Commercial OODBS's

Existing commercial products such as GemStone, Vbase and VISION will continue to undergo refinements for performance. I see the most need for work in associative access support for bulk data types, and optimization techniques that take advantage of such support [GM]. Other added features will be application development environments and ways to modularize object spaces.

ACKNOWLEDGEMENTS

I always try to steal my ideas from the best sources. Sources of my thievery that come to mind are Earl Ecklund, Stan Zdonik, Francois Bancilhon, Jacob Stein, Goetz Graefe, Lougie Anderson, Yann Viemont and Jim Ravan. Dave DeWitt and Maggie Eich offered comments (some of which were accepted) on a draft of the paper. This work was partially supported by NSF award IST 83 51730, co-sponsored by Tektronix Foundation, Intel, Digital Equipment, Servio Logic, Mentor Graphics and Xerox.

7. REFERENCES

[ABC+]
M. P. Atkinson, P.J. Bailey, K. J. Chisholm, W. P. Cockshott, R. Morrison. An approach to persistent programming. *Computer Journal 26:4,* November 1983.

[ABD+]
M. Atkinson, F. Bancilhon, D. DeWitt, D. Maier, S. Zdonik. The object-oriented database system manifesto (a political pamphlet). Working paper, November 1988.

[AH] T. Andrews and C. Harris. Combining language and database advances in an object-oriented development environment. *Proc. of the Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications,* Orlando, Florida, October 1987.

[Appin]
Proc. of the Appin Workshop on Persistence and Data Types, Appin, Scotland, August 1985. Persistent programming research report 16, Univ. of Glasgow.

[BBB+]
F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard, F. Velez. The design and implementation of O2, and object-oriented database system. In [OODB88].

[BBG+]
D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twitchell, T.E. Wise. GENESIS: An extensible database management

system. *IEEE Trans. on Software Engineering 14:11,* November 1988.

[BC+] J. Banerjee, H.-T. Chou, J. F. Garza, W. Kim, D. Woelk, N. Ballou, H.-J. Kim. Data model issues for object-oriented applications. *ACM TOOIS 5:1,* January 1987.

[Be] A. J. Berre. SOOM and Tornado-*: Experience with database-support for object-oriented applications. In [OODB88].

[BM] F. Bancilhon and D. Maier. Multilanguage object-oriented systems: new answer to old database problems. *Future Generation Computer II,* K. Fuchi and L. Kott (eds.) North-Holland, 1988.

[BO] P. Buneman and A. Ohori. Using powerdomains to generalize relational databases. To appear *Theoretical Computer Science.*

[BZ] T. Bloom and Z. Zdonik. Issues in the design of object-oriented database programming languages. *Proc. of the Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications,* Orlando, Florida, October 1987.

[Ca] L. Cardelli. A semantics of multiple inheritance. *Intl. Symposium on Semantics of Data Types,* Sophie-Antipolis, France, June 1984. Lecture Notes in Computer Science 173, Springer-Verlag.

[CD+] M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, J.E. Richardson, E.J. Shekita, M. Muralikrishna. The architecture of the EXODUS extensible DBMS. In [OODB86].

[CDP] M.J. Carey, D.J. DeWitt, S.L. Vandenberg. A Data Model and Query Language for EXODUS. Univ. of Wisconsin, Madison, Computer Sciences TR 734, December 1987.

[CW] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys 17:4,* 1985.

[CS] M. Caruso and E. Sciore. Contexts and meta-messages in object-oriented database programming language design. *Proc. ACM-SIGMOD International Conference on Management of Data,* Chicago, Illinois, June 1988.

[Da+] P. Dadam, et al. A DBMS prototype to support extended NF2-relations: An integrated view on flat tables and hierarchies. *Proc. ACM-SIGMOD International Conf. on Management of Data,* Washington, D.C., May 1986.

[DBE] *Database Engineering Bulletin 8:4* Special Issue on Object-Oriented Systems, F. Lochovsky (ed.), December 1985.

[DBPL] *Proc. of the Workshop on Database Programming Languages,* F. Bancilhon and P. Buneman (eds.), Roscoff, France, September 1987.

[DGL] K. R. Dittrich, W. Gotthard, P. C. Lockemann. DAMOKLES—A database system for software engineering environments. *Proc. IFIP Workshop*

on Advanced Programming Environments, R. Conradi, T. M. Didriksen, D. H. Wanvik (eds.) Trondheim, Norway, June 1986, Lecture Notes in Computer Science 244, Springer-Verlag.

[EE+] D. Ecklund, E. Ecklund, R. Eifrig, F. Tonge. DVSS: A distributed version storage server for CAD applications. *Proc. VLDB XIII,* Brighton, England, September 1987.

[Fi+] D. Fishman, et al. Iris: An object-oriented database management system. *ACM Trans. on Office Information Systems 5:1,* January 1987.

[Fo+] S. Ford, et al. ZEITGEIST: Database support for object-oriented programming. In [OODB88].

[GM] G. Graefe, D. Maier. Query optimization in object-oriented database systems: a prospectus. In [OODB88].

[HK] S. Hudson and R. King. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. To appear *ACM Trans. on Database Systems.*

[HL] R.L. Haskin and R.A. Lorie. On extending the functions of a relational database system. *Proc. ACM-SIGMOD International Conference on Management of Data,* Orlando, Florida, June 1982.

[HM] M. Hammer and D. McLeod. Database description with a semantic data model: SDM. *ACM Trans. on Database Systems 6:3,* September 1981.

[JF+] D. Jagannathan, B. L. Fritchman, R. L. Gluck, J. P. Thompson, D. M. Tolbert. SIM: A database system based on the semantic data model. *Proc. ACM-SIGMOD International Conference on Management of Data,* Chicago, Illinois, June 1988.

[KB] M. A. Ketabchi and V. Berzins. ODM: A object-oriented data model for design databases. Univ. of Minnesota Institute of Technology TR 85-41, October 1985.

[Li] B. Liskov. Data Abstraction and Hierarchy. *Proc. of the Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications: Addendum,* Orlando, Florida, October 1987.

[LMP] B. Lindsay, J. McPherson, H. Piradesh. A data management extension architecture. *Proc. of the ACM-SIGMOD International Conference on Management of Data,* San Francisco, California, May 1987.

[Lo] D.B. Lomet. A data definition facility based on a value-oriented storage model. *IBM Journal of Research and Development, 24:6,* November 1980.

[Ma] Making database systems fast enough for CAD applications. In *Object-Oriented Concepts, Applications and Databases,* W. Kim and F. Lochovsky (eds.), 1989.

[MD] F. Manola and U. Dayal. PDM: An object-oriented data model. In [OODB86].

[MER] F. Mariategui, M. H. Eich, S. Rafqi. The object oriented data model defined. SMU TR, March 1988.

[MSOP] D. Maier, J. Stein, A. Otis, A. Purdy. Development of an object-oriented DBMS. *Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications,* Portland, Oregon, September 1986.

[O'B] P. O'Brien. Common object-oriented repository system. In [OODB88].

[OH] S. L. Osborn and T. E. Heaven. The design of a relational database system with abstract data types for domains. *ACM Trans. on Database Systems 11:3,* September 1986.

[OODB86] *Proc. 1986 International Workshop on Object-Oriented Database Systems.* K. Dittrich and U. Dayal (eds.), Pacific Grove, California, September 1986.

[OODB88] *Advances in Object-Oriented Database Systems: Proc. 2nd International Workshop on Object-Oriented Database Systems,* K. Dittrich (ed.), Bad Muenster am Stein-Ebernburg, West Germany, September 1988, Lecture Notes in Computer Science 334, Springer-Verlag.

[POS87] *Workshop on Persistent Object Systems: Their Design, Implementation and Use.* Appin, Scotland, August 1987. Persistent programming research report 44, Univ. of Glasgow.

[POS89] *Workshop on Persistent Object Systems: Their Design, Implementation and Use.* Newcastle, Australia, January 1989.

[PS+] H.-B. Paul, H.-J. Schek, M. H. Scholl, G. Weikum, U. Deppisch. Architecture and implementation of the Darmstadt database kernel system. *Proc. ACM-SIGMOD International Conference on Management of Data,* San Francisco, California, May 1987.

[RKC] W. B. Rubenstein, M. S. Kubicar, R. G. G. Cattell. Benchmarking simple database operations. *Proc. ACM-SIGMOD International Conference on Management of Data,* San Francisco, California, May 1987.

[RMS] S. Riegel, F. Mellender, A. Straw. Integration of database management with an object-oriented programming language. In [OODB88].

[SS] T. Sheard, D. Stemple. Automatic verification of database transaction safety. To appear, *ACM Transactions on Database Systems.*

[St86]. M. Stonebraker. Inclusion of new types in relational database systems. *Proc. Second IEEE Data Engineering Conf.,* Los Angeles, California, February 1986.

[St87] M. Stonebraker. The design of the POSTGRES storage system. *VLDB XIII,* Brighton, England, September 1987.

[SR] L. Rowe and M. Stonebraker. The POSTGRES data model. *VLDB XIII,* Brighton, England, September 1987.

[TL] D.C. Tsichritzis and F.H. Lochovsky. *Data Models.* Prentice-Hall, 1982.

[WF+]D. Weinreb, N. Feinberg, D. Gerson, C. Lamb. An object-oriented database system to support an integrated programming environment. *Database Engineering Bulletin 11:2,* June 1988.

[WS+]P.F. Wilms, P.M. Schwartz, H.J. Schek, L.M. Haas. Incorporating data types in an extensible database architecture. IBM Almaden Research Report RJ 6405, August 1988.

[WZ] P. Wegner and S. Zdonik. Language and methodology for object-oriented database environments. *Proc. of the Nineteenth Annual International Conference on System Sciences,* Honolulu, Hawaii, January 1986.

[Za] C. Zaniolo. The database language GEM. *Proc. ACM-SIGMOD International Conf. on Management of Data,* San Jose, California, May 1983.

[ZM] J. Zhu and D. Maier. Abstract objects in an object-oriented data model. *Proc. of the Second International Conference on Expert Database Systems,* Tysons Corner, Virginia, April 1988.