# The FOOD Object-Oriented Database*

Erik Odberg and Svein Erik Bratsberg†
Division of Computer Systems and Telematics
Norwegian Institute of Technology

Presented at DND Seminar on Object-Orientation, Geilo, Norway April 23, '91

March 31, 1991

**Abstract**

In this paper, we present the FOOD Object-Oriented Database. This is done in the perspective of a general framework and motivation for Object-Oriented Databases. FOOD is fully Object-Oriented, essentially providing persistency for C++ objects. Additional features extending the C++ data model include explicit relations, versioning using the Change-Oriented Versioning mechanism, a trigger concept for having some operations *implicitly* executed and persistent iterators for manipulating *sets* of objects. Many of these are features not regarded mandatory in an OODB environment, but we have felt these as valuable for fulfilling the requirements from the application areas addressed by FOOD. These areas are characterized by having long design transactions operating on complex object structures. The paper emphasize an overall presentation of the FOOD model in a general OODB perspective, omitting detailed discussions of the individual features.

## 1 Introduction and Motivation

In the recent years, there has been a growing understanding in several communities that classical models for database storage are not sufficient anymore. In principle, this is related to two particular issues:

- Modeling power
- Efficiency

Very many application areas pose requirements for database support which may not be satisfied by relational, hierarchical or network models, which are inherently *record-oriented*.

Application areas which have experienced these problems, are as diverse as CAD/CAM, Office Information Systems, Artificial Intelligence (AI), Software Engineering Environments (SEE), Geographical Information Systems, image processing and various other engineering and design application areas. These areas are typically characterized by handling *complex objects*, i.e. objects with an inner structure rather than record-like structures. The deep semantics of each application object should be modeled and handled by the database system.

Many people have experienced problems with the relational model. The modeling features are, for many application areas, seen as highly inappropriate. There is no provision for posing advanced queries to the database as could have been possible within ordinary programming languages. Thus,

*This work has been been done in affiliation with the EPOS - Expert system for Program and ("Og") System Development project, which lasted from 1986-1990. EPOS was supported by the Royal Norwegian Council for Scientific and Industrial Research (NTNF) through project grant ED0224.18457.

†Detailed address: Division of Computer Systems and Telematics (DCST), Norwegian Institute of Technology (NTH), N-7034 Trondheim-NTH, Norway. Phone: +47 7 594484. Fax: +47 7 594466. Email: eriko@idt.unit.no

a comprehension has arose that far more concepts from the programming language world are needed within the database systems.

Today, many engineers solve the problem in an ad-hoc way. The typical solution for CAD/CAM systems is to load the complete complex structure into main memory upon session start, and store it back into files when the session is completed. This have the advantage of taking the full benefits of the programming language features: Rich expressibility and complex structures, as well as being able to operate on the data using all the functions and operators available for modern programming languages.

This solution is still undesirable, because there is no proper provision for concurrent use of the data structures. The work is inherently single-user oriented. For many engineering application areas this is a significant problem, as the size and complexity of the problem situations often require that more than one single person is allocated on the job. Also, fetching the complete workspace structure may take a significant amount of time, even if only parts of it is really interesting. That is, proper database functionality is required in order to have a proper working solution.

Some people tried to map the complex data structures onto classical model database systems, in order to solve the modeling problem. This had some advantages, but the underlying technology imposed the same problems as before with respect to efficiency: The classical model systems are not appropriate for representing the complex structures, and thus a mapping process will imply no improvement in performance.

In summary, what is needed is to merge the concepts of programming languages with those of the databases. There is an obvious need for providing both within the same framework. Both the programming language people and the database people have recognized the same needs for new functionality.

To solve the performance problem, new basic technology for storing the objects is required as well. It is an important goal to unify the modeling structures with the actual storage structures, so that a natural and simple mapping between the two worlds is provided.

## 1.1 The introduction of OO principles

Object-oriented principles are known for providing good support for problems like those described above, and since 1984 we have seen an increasing amount of research prototype and commercially available Object-Oriented Databases (OODBs) appear. OO as a design and implementation paradigm, is well-known for

- abstracting the essentials, through the distinction between interface and implementation/-representation. This provides better understanding.
- Encapsulating data and oeprations to work on the data in one paradigm: The object.
- easening reuse of code and specification, through inheritance and subtyping.
- providing a more natural modeling paradigm, better reflecting many real-world phenomena.
- fomalizing the interfaces better.

OO programming languages first appeared embedding this new paradigm. However, in the recent years there has been a growing interest in applying OO to the *complete* life cycle, performing OO analysis, OO design, and finally implementing this using an OO language. OODBs follows as a natural result of this, giving rise to an implementation environment for databases within "inherently" OO problem domains.

We do *not* claim that OODBMS is the definite solution to all database problems. The classical models, in particular manifested by the relational model, is found to have serious shortcomings for *certain application areas*. The OODBMSs emerging are believed to be a fruitful direction to follow in order to try to solve the particular problems as described above.

However, it is a well-known problem that database applications are often designed with respect to what is *available*, and not necessarily taken into concern what is most suitable. According to

this, we might expect that many "classical" application areas will find the use of new technology, represented by OODBMSs, to be valuable.

Thus, in conclusion, OODBs:

- provide better modeling features, by supporting a more natural representation of many real-world phenomena. New design methodologies such OO analysis (OOA) and design (OOD) may be mapped more directly onto a database specification.
- allow for efficient storage and retrieval of complexly structured and heterogeneous objects.
- support reusability of code and specifications.
- ease the maintenance phase, by providing better understanding.
- ensure better database consistency, by accessing database structures through defined interface methods, only.

The rest of this paper is organized as follows: Section 2 presents the basic concepts of OODBs, Section 3 describes the FOOD data model, Section 4 presents the interface towards the database, Section 5 is an introduction to evolutionary aspects of OODBs, Section 6 is a brief description of the FOOD versioning model, Section 7 proposes a scheme for versioning of types, while Section 8 concludes the paper.

The paper is a revised version of [OB90].

# 2 What is an OODB?

In this section, we will briefly describe the common consensus of an OODB, as has been identified in [ABD+89]. This taxonomy, which may seem conservative related to many of today's OODBs, will be contrasted to the concepts as included within FOOD:

The mandatory features are:

**Object Identity** Objects are identified by a unique, system-defined identity, rather than by value.

**Types and Classes** A concept of type (intensional aspects), or class (implementational aspects) should be provided. Many OO systems blend these two issues.

**Encapsulation** It should be distinguished between an object's interface to external clients, and its implementation. Another interpretation is that data and operations on the data are bundled together in one abstraction.

**Complex objects** Objects may be composed of other objects.

**Inheritance and subtyping** A class may *inherit* properties from other classes, possibly redefining some of the properties. Subtyping is the principle that objects of a type may be used anywhere that a supertype object is expected. These principles allow for reuse of code and interface, as well as precise modeling. Multiple inheritance is defined as an optional feature.

**Late binding** Binding of a method invocation is delayed till runtime, taking into concern the type of the object having the method invoked, rather than the type of the pointer.

**Computational completeness** Any computable function should be possible to express through the database interface language.

**Extensibility** It should be possible to add user-defined types, and not be limited to a predefined set.

**Persistence and the basic DB Amenities** Concurrency, Recovery, Rollback etc.

**Ad-hoc query language** There should be a simple, interactive way of getting access to the data, in a reasonably declarative way.

Some optional features are:

**Transactions** Support for long non-serializable (design) transactions.

**Versioning** of objects. Objects may exist, and should be possible to access, in multiple versions reflecting some evolution of the object.

**Schema Evolution** Support for making changes to the database schema, with proper manipulation of already existing objects.

**Integrity constraints** The ability to specify legal combinations of values.

**View definitions**

**Distribution**

Dittrich [Dit88] made a distinction between *structurally*, *behaviorally* and *fully* OODBs, denoting DBs providing complex objects, behavior[1] and both of these, respectively. Today's trend is to require an OODB to be fully OO, according to the definitions as presented above.

Standardization work is currently on the way, in order to make a common taxonomy of what is required from an OODB. Both OMG - Object Management Group, and ANSI OODBTG - OODB Task Group are actively trying to reach consensus about the matter, and are expected to reach conclusions in a short time.

It is also worth mentioning another direction for extending the database functionality [Sto90]. In this paper, it is argued for that existing relational technology may be used as a basis for adding new features, trying to solve the same problems as those for OODBs. Many of the features have direct correspondents within [ABD+89]. A bit of a war between these two manifests is being fought.

# 3 The FOOD Data Model

The data model is fully object-oriented, allowing for stating complex object structures and type-specific operations on objects. FOOD provides support for C++ objects, and adopts a concept of explicit relations in the model. Manipulation of sets of objects is regarded important, and is supported by having a concept of iterator. Iteratos are persistent as other objects in the database. FOOD supports Change-Oriented Versioning [LCD+89], and proposes a scheme for versioning of types similar to Encore [SZ86], [Zdo86]. Many of these extends considerably beyond the basics for OODBs as described in [ABD+89].

## 3.1 Types, Classes and Objects

FOOD distinguishes between *Types* and *Classes*, in a way a bit different from [ABD+89]. Types describe the properties of the objects of the type, consisting of *attributes*, *operations* and *containment relations*. These concepts, which are commonly denoted *properties*, are elaborated upon below.

FOOD allows for defining *subtypes* of other types, inheriting properties from one or more supertypes. Objects of subtypes may occur where objects of any supertype is expected. Possible conflicts for properties due to this multiple inheritance implies that *qualification* is made. *Inheritance* is related to subtyping in the same way as for C++.

Types are specified in a *schema*, using a DDL which is a simplified C++, adding capabilities for specifying explicit *relation types*. The schema is compiled into proper C++ class definitions, which are included in the application programs. These C++ classes cater for transparent access to persistent objects in the database[2].

In FOOD, a Class is the *extension* of a type, i.e. a container for all the instances of the corresponding type, including all subtypes[3]. A Class is itself an object (implicitly created), and may have

[1] Behavior implies that the model supports for associating functions with objects.
[2] As will be shown later, relations types, which is no explicit concept in C++, are treated specially.
[3] That it, we have inclusion semantics on the classes.

attributes associated with them. These attributes are accessed from the objects as ordinary object attributes, but are shared among the instances in the same way as *static* data members in C++. There are special operations to query the class object.

The C++ visibility declaration capability has been adopted by FOOD, so that each property has one of the following visibilities:

**Private** : The property is internal to the object only, and may not be accessed directly by the external environment, only through *public* operations defined.

**Public** : The property is directly accessible by the external environment (other objects). In order to achieve the usual encapsulation principles of distinguishing implementation from specification, attributes will most often be declared private and operations public.

**Protected** : The property is private to the external environment, but will act as public with respect to subtypes of the actual type. That is, they may be invoked from operations in subtypes.

Public attributes for a supertype will be public for subtypes as well. However, as in C++ there is a possibility to specify that all inherited propertied should be made private for the type which inherits.

Attributes in FOOD are typed and named, as in C++. A basic set of attribute domains is defined, together with a few type constructors to create higher level domains such as sets, bags, enumerations and arrays.

A particular attribute domain in FOOD is the *longfield*. A longfield is the database representation for files, abstracting the actual UNIX file. There are predefined operations defined for manipulating longfield attributes, the most interesting being the ability to copy the attribute contents out to a file and into the database. Thus, it is a way to include files transparently as part of the database, ensuring appropriate locking.

## 3.2 Operations

Operations is the mechanism used for specifying object behavior, and take the appearance of C++ functions. The function name and signature is specified in the schema, the implementation of these are stated on a separate file and compiled into the server[4].

For every attribute defined for a type, protected operations to set and get the persistent value of the attribute will automatically be defined. All the database object access will be performed through calls to these operations, which are utilized by the publicly defined functions. That is, they are visible to implementors of operations for a class, but not externally unless explicitly stated so. In this way, it is possible for the application programmer to have complete transparency with respect to access to the persistent objects. The only operations which are visible will be those defined by the database designer.

Consequently, FOOD operations are really ordinary C++ operations which may utilize the FOOD database interface functions and the attribute access functions as well. It should be noted that user-defined operations may not be associated with class objects. Visibilities are associated with operations as for other properties.

## 3.3 Relations and relationships

The notion of relationships is another controversial issue for OODB systems. [ABD+89] does not consider this issue at all, but some OODB systems support a notion of relationships between objects. The fact that many OODBs completely lack a proper concept of relationship, has been used as an important argument against OODBs by the (extended) RDBMS community.

[4] For several reasons, we have taken the approach to compile the operations into the server, rather than the applications.

In our opinion, a proper concept of relationships is necessary in order to provide a natural modeling construct, in addition to objects. This is more than just special objects having object references as a special attribute domain. There are several reasons for introducing this concept:

- It is a very useful modeling construct, known from the ER-model, which clearly states a conceptual difference between objects and relationships between objects. Thus, FOOD brings the implementation domain closer to the modeling domain.
- They are symmetric, as opposite to one-way references known from C++.
- Attributes which are conceptually to be regarded as belonging to some association, need not be attached to some object.
- It avoids incorrect implementation of relationships. Conceptual relationships are often needed in an application, and FOOD provides this facility as part of the model. For instance, dangling references is not possible.

The FOOD relationships have the following properties:

- Relationships have no identifier – they are not objects! Thus, we may say that the instances are identified by their contents, and thus the main concept of the relational model is integrated within the model. We feel proper relationships are something completely different from objects identified by a unique ID, a feeling inherited from the ER model. A relationship is identified by the set of roles and the associated attribute values.
- Relationships are n-ary associations between objects, called the *roles*, allowing for attaching attribute values to the actual relationships. No other properties are associated with the relationships.

  Each role has associated a *cardinality constraint* with it, stating the maximum cardinality for this role. This cardinality may be *one* or *many*. There are no lower cardinalities, thus we may not specify existence dependencies.
- Relationships are instances of *Relation types*, describing the relationship properties. Relation type may be subtyped as well, as explained below.
- Relationships are accessible through the *Relation* – the extension of the relation type. This is the container for all the instances of the relation type (as well as all the subrelation types), and is implicitly defined for each relation type. Thus, the Relation parallels the Class for ordinary types. There are predefined, globally visible operations to access the relationships as contained within the Relation object. No user-defined operations are possible.
- FOOD relationships are *ordered*. I.e., users may impose specific sequences between the relationships within a Relation, if this is desired. For instance, if there is a relation type between book and chapter types, users may insert the chapters in this book in a specific sequence, in such a way that upon traversal through the relation the same order will be implied.

Relation types may be *subtyped* into more specialized types, in a way similar to for ordinary types. I.e., a relationship being an instance of a subrelation type, may occur wherever any type instance is expected. Also, properties of supertypes are inherited. The following subtyping operations are defined:

- Constrain the cardinality
- Subtype the roles
- Add new attributes

FOOD does not support *multiple* inheritance for relation types, as there are some problems in defining meaningful semantics for this.

There are functions implicitly defined for Relation objects to *traverse* through the contents of the relation. In fact, as the relationships are not objects, the only way to access them is through

accessing the relation object. For each relationship which is returned, it is possible to access the attribute values as associated with this relationship.

User may instantiate different *threads* through the relation. These are really just placeholders for different *current relationships* in the iterator. These threads are necessary in order to traverse through three-like (recursive) structures of relationships in the same Relation. The threads are local to the application.

### 3.3.1 Containment relation types

A special kind of relation type in FOOD is the *Containment Relation Type*. Instances of these have the following characteristics:

- They are binary, i.e. only two roles.
- Special semantics with respect to deletion and copying, so that the complex object is treated as one object.
- Cardinality constraints on the roles may be used for specifying *shared subobjects*.

Containment relation types is the FOOD mechanism for specifying *Complex objects*. These are also the only relation types being specified *within* the ordinary types. The reason for this, is that subobject containment is felt to be a an intrinsic property of the objects, rather than external to the objects participating. Consequently, there is no explicit relation object for containment relation types: The relationships are contained within the class of the containing object type. Thus, the operations to manipulate these relationships are contained in the class of the containing type.

## 3.4 Iterators

FOOD supports a concept of persistent iterators, being containers for a collection of objects from a particular class. These iterators are objects of the predefined type Iterator, and there is a predefined set of operations to manipulate these objects[5]. Notably, these are operations to traverse through the objects in the iterator, to insert objects at some place in the iterator, or to delete them from the iterator[6].

Iterator objects are created and deleted similar to ordinary objects. Iterators may be *instantiated* through a specification language[7], but objects may as well be inserted manually one by one.

For each ordnary type, we may create an arbitrary number of iterators, and each object may be member of an arbitrary number of iterators (for the class).

Iterators are convenient in order to operate on a subset of objects in a class, possible as selected as the result of an associative retrieval. In order to have user-defined properties for iterator objects, it is possible to specify *subtypes* of Iterator adding new operations in addition to the ones specified in Iterator. For instance, operations to manipulate upon *all* objects in an iterator, may be defined. Thus, the iterator concept as provided by FOOD is a basic mechanism for providing support for defining further semantics. Also the concept of iterators is very convenient to combine with an interactive query language towards FOOD.

The concept of iterators like this one is not common within OODBs today.[ABD+89] does not specify this at all, [Sto90] emphasizes that the collection contents should be possible to specify in more than one way.

[5] As iterators contain objects in specified classes, there are implicitly defined *subtypes* of Iterator for each type in the schema.

[6] Note that deleting an object from an iterator is completely different from deleting the object.

[7] This language is yet to be defined, but it should be able to perform associative access to the objects. There will be no provision for this instantiation criteria to dynamically hold for the objects part of the iterator.

### 3.5 Triggers

Stonebraker [Sto90] has identified the importance of having some *rule mechanism* in the next generation of DB systems. This is neglected by [ABD+89], but in FOOD we have included a notion of a *Trigger* in order to support this functionality.

Triggers in FOOD are operations which are only *implicitly* executed, as the result of some database or external *event*. These events are associated with system- and user-defined database operations, or may be explicitly defined through an application signal or time. Also included within the trigger framework, is the notion of a *condition*, which evaluates whether the actual trigger operation is to be executed or not. The condition is an arbitrary C++ condition. Also, trigger operations have the same appearance as any other FOOD operations. They are specified on separate files along with the ordinary database operations.

Triggers may be associated both with ordinary objects and classes, and are very convenient to apply for ensuring semantic consistency and user-defined propagation.

The possible events, are:

- General database event (the collection of all other events).
- Database operations (that are not attached to objects).
- The collection of all user-defined events.
- User-defined events. There are three different operations to raise these events. These three operations correspond to an immediate event, a relative time event, and an absolute time event. These are "operations" that are not database events.
- Attribute modification on a class attribute.
- Invocation of class object operations.
- Attribute modification on an object attribute.
- Invocation of object operations.

The events are organized in a hierarchy, used for checking the conditions.

## 4 The Database Interface

There are many different principles for interfacing to an OODB. In FOOD, we have emphasized the use of C++, and thus made database access as transparent as possible for C++ application programmers:

- C++ classes are created based on the schema specification, and are used by applications to manipulate persistent objects transparently. In this way, access to persistent objects is, from application programs, made in the same way as access to ordinary objects, but the underlying semantics is much more complex.
- Class and Relation objects are naturally represented the same way.
- Database objects are accessed through application C++ objects representing the persistent equivalent.
- The DDL language is a simplification of C++ class definitions.
- Operations associated with each object type are ordinary C++ member functions.

In order to get a handle into the database structures, symbolic object identifiers representing the object classes will automatically be defined. Thus, initial application access to the objects must be done using these symbolic identifiers, and create the in-memory C++ representations.

Many OODBs also have an *ad-hoc query language* towards the database objects. This is identified by [ABD+89] as well. Most systems envisage an SQL-based language for the ad-hoc handling of objects. Such a language, OSQL, has also been defined for FOOD [Bra90].

We are now in progress of defining a new Ad-hoc interactive language for FOOD, which better reflects the OO paradigm. We believe SQL-based languages are not very well suited for catching all the semantics of highly complex object structures. Thus, we will try to design an interactive language which reflects OO principles in general and the special FOOD concepts in special, but adds associative query capabilities. As OO is a way of thinking, more than special syntax, we think it is important that the language reflects an OO view.

The matter of object query languages was recently discussed in a panel session at OOPSLA '90 [Mei90]. There was agreement that *both* imperative *and* declarative languages, preferably integrated, may be advantageous in an OODB, solving different classes of problems.

# 5 Evolutionary aspects

One of the important characteristics of problem domains which are typically targeted by OODBs, is the evolutionary nature. Specifications and definitions frequently change and evolve during a design process. This applies to object instances in the database, as well as the *descriptions* of the objects (the types) in the database. Consequently, there is a very important need for supporting such activities by the database.

In principle, we may identify the following categories of evolution:

- Evolution of objects. Objects may evolve, and we are often interested in recording old states of objects. This gives rise to a concept of object versioning.
- Changing the specification of a type. Typically, old specifications need no longer conform to new requirements, and a desire to change the types system (the schema) may arise. This is inherently a very complex matter in the general case.
- Changing the behavior of specific objects. Often, a specific object in the database conceptually is promoted to have some new behavior, or losing existing behavior. Still, it should be regarded the same object. This gives rise to changing the *type* of an existing object, to reflect real-world evolution. For instance, a child will grow to become a student, the student becomes an employee, and eventually retire. This describes an evolutionary process, and something which is properly modeled by supporting a concept of changing the type of an object.

All these are supported by FOOD, and the concepts are orthogonal to each others. We will discuss the FOOD approach to the first two of these in the next sections. Also, FOOD provides special operations to *change* the type of a specified object.

# 6 Versioning of Objects

Interestingly enough, [ABD+89] does not consider the issue of having a versioning mechanism as mandatory. No attempt is made to state a common framework for handling versioning of objects. This naturally reflects the lack of consensus within the OODB world, resulting in many different proposals. In our opinion, and has have been stated above, versioning is essential for many of those application areas which are typically targeted by OODBs. Consequently, a versioning mechanism is central in the FOOD data model.

FOOD applies *Change-Oriented Versioning* (COV) [LCD+89] as the basic mechanism for handling versioning of database objects. This paradigm is distinguished from the most widely applied *Version-Oriented Versioning* (VOV) model, in the following ways:

- Versioning is global to the database, rather than to a particular object. This means that a complete database version is chosen by one version specification, rather than selecting a configurations as consisting of specified versions of each individual object.

- The emphasis is on selecting the database as a *consistent set of functional changes* to the database, rather than as some collection of individual object versions which in principle need not be consistent. The notion of the functional change is the primary concept, thus the database version is selected as having a specified set of functional properties. In VOV, the actual version is primary, the change playing a secondary role as being the difference between two versions.
- COV is strong on supporting variant-like structures, as the concept of change propagation is important. This feature is often difficult to handle properly for VOV.

For these reasons, we have a strong belief that COV is better suited for versioning of complexly evolving object worlds, more naturally reflecting the nature of the evolution. The notion of a functional change is believed to better reflect the user's notion of the differences between two versions. Also, we believe that the complete database state is important for the versioning, as picking versions of the individual objects need not properly reflect the relationships between the objects in the database.

Within the EPOS project a full implementation of COV for an extended ER database environment is now in progress [Lie90], [Mun90]. The emphasis is on supporting complexly evolving software. We await the results from functional and performance-related tests using this database, before we will be able to conclude about the applicability of the model. The initial experiments and results are promising, and described in [LHC90], but more realistic experiments must be made and the efficiency problem need to be solved before some proper conclusions about the applicability may be made.

Closely related to the versioning model, is the notion of a *long transaction*. This must be distinguished from (short) transactions known from the conventional DBs, which are mechanisms for defining a collection of DB operations to be regarded as one unit, and thus all the operations should completely done in order to maintain a consistent DB.

A long transaction is a concept having arose from many design and engineering environments, where a typical way of working is to *check out* a part of the database for exclusive access, perform some changes to this, and check it back into the database afterwards. Thus, this mechanism is an alternative to ordinary concurrency control issues.

In FOOD, the long transaction concept is the basis for creating new database versions. A long transaction is initiated by stating an *ambition* for the changes to be done. This is a logical expression stating the *propagation* of the changes made within the long transaction. I.e., for which database versions should the changes be applied. Upon committing the long transaction, the changes made are made visible within this scope.

It should be noted, that COV is applied both to the ordinary database objects, as well as to the individual longfields ("files") as being attributes on objects. For relationships, which have no identity, each relationship either is or is not visible within a particular database version.

For a more elaborate presentation of COV and the COV concepts than we are able to include here, it is referred to [Lie90], [LCD+89], [Hol88].

# 7 Changing the Type System

Design decisions are frequently changed, and database specifications (schemas) may be required to be changed along with these. This has been recognized as a huge problem using conventional database systems but is very important for areas targeted by FOOD. Many changes which are really needed, are not performed due to lack of appropriate technology. Consequently, there is an increasing interest in this matter as related to to the introduction of OODBs.

The concept of type *evolution* is briefly mentioned in [ABD+89] as a feature for which no consensus is reached. As far as we know, for efficiency reasons no commercial products presently supports it properly, but some prototyping work has been made.

In this paper, we will distinguish between:

- **Type evolution**, which will support for schemas to evolve, but there may only be one current version at a time, so that all objects and application programs must conform to this latest version of the schema. I.e., existing objects must be converted in some way, eagerly (the *conversion approach*, e.g. GemStone [PS87]) or lazily upon first-time reference [BH89]. [LH90] describes an interesting mechanism for making this conversion, the conversion procedures properly reflecting the semantics of the actual modification.

  Another possibility is not to reorganize physically the database structures, but provide an interface from new definition to old representations, interpreting the contents upon each access (the *screening approach*), e.g. as is done for ORION[8] [BKKK87][9].

- **Type versioning**, which will support proper versioning of types, so that application programs and objects may conform to different versions of the schema. Encore/Observer [SZ86], [Zdo86] supports multiple type version views towards a database where the objects of the same type may have different representations. Regrettably, they have to our knowledge no prototype implementation to demonstrate.

Some systems evade the problem by allowing for making new versions of schema types, but without having any possibility to regard old objects as instances of the new definition, or vice versa. In this way, there is no relationship between the versions of the type, apart from the fact that they may have some common implementation. We do not regard such a solution in this section.

No matter what scheme is chosen, there may be limitations to what class of schema changes may be allowed. E.g. ORION will only support for attribute *generalizations*.

In principle, there are two main important problem areas related to versioning of schema types:

- **The structural problem**: Catering for the objects on basic storage to have a proper format, irrespective of version. Usually, a conversion of all the objects in the database is performed in order to accomplish this.
- **The behavioral problem**: Catering for the consistent handling of objects possible having different implementation and/or behavioral specification (interface).

We feel the type evolution approach is not completely satisfactory, for the following reasons:

- FOOD is, through supporting COV, particularly concerned about supporting *variant-like* development structures. This should be reflected in the type level as well, by supporting a scheme of multiple interfaces towards the database objects. The general implications of this will be invalidated if the objects are converted to *one* type, the latest.

  There may be a need for the more powerful variant-like handling of the schemas being modified.

- Conversions may imply information loss.
- The conversion, being either eager or lazy, may be expensive to perform.
- All the applications may be required to be updated and recompiled in order to reflect the last version of the schema.

## 7.1 The Model

FOOD proposes a scheme of proper type versioning, allowing for objects of the same type in the database to have different appearance, corresponding to different versions of the type. In addition, different application programs may view the objects differently according to the type version they (the applications) conform to. A particular application may be attached to any version of the schema, and may change to another version if this is desired. However, this will

[8] ORION is now commercialized under the name ITASCA.

[9] It should be noted that neither of these schemes are implemented in the products which are now being sold, partly due to efficiency issues.

require recompilation and possibly other changes to the application program. The unit of change is at the *schema* level, not the type level, as we feel there are strong relationships between different types in the same schema, as for instance with respect to where in an inheritance hierarchy to place some information. Type versioning is completely orthogonal to ordinary object versioning.

The changes to the schema types may have two appearances, being external (change to the interface) or internal (change to the operation implementation or collection of attributes defined). Also, the type inheritance structure may change, having quite serious implications to the database.

The problem may be the following:

- An object may not be able to respond to an operation defined for the application version of the type, as the operation is not defined for the objects type version.
- An operation may fail to execute, as it references properties of the object which are non-existent. This requires *reimplementation* of the relevant method for the object's type version. It is important to note that the operations are really executed in the context of the database, and are not compiled into the application program.

The idea is to introduce *error handlers* each time a change is made to a type in the schema. These error handlers are operations trying to cover the functionality which is not supported by some type versions, as related to the other versions. Thus, if some object may not respond to some operation call because the operation is not defined for the object's type version, the error handler defined will be called. The error handlers are specified by the users, may access the internals of the objects, and in this way are very similar to the ordinary operations. The semantics of the error handlers are completely defined by the user.

Proper versioning of types may have serious implications to the total functionality of the database, and thus the scheme must be used with care. Some changes may seriously redefine the objects' appearance, having big implications to the application programs as well. Also, there will be some constraints to which changes are legal, due to the fact that no meaningful semantics may be defined.

It must be noted that this scheme introduces one more level of indirection, and thus is bound to imply some degradation in performance. However, we will work more within the field of type versioning in the future. Also, the matter is far more complex than what may properly be described in the available space. We feel that much more work is needed within the handling of schema modifications in general, as this is certainly a very difficult issue to handle properly without serious performance degradations. But, as have been described, the functionality may have sincere benefits for database designers and application programmers.

# 8 Status, Conclusions and Further Work

A limited functionality prototype has been implemented, intended for providing further insight into the problem area as well as clarifying thoughts, demonstrating functionality and performing some initial performance measures. The prototype is based on C-ISAM as underlying persistent storage, a solution which we have mostly good experiences with. The "hard" issues, such as versioning of instances and types, has not been prototyped. COV is currently being implemented and tested within the EPOS database, which essentially conforms to an ER model extended with inheritance.

The work on and experiences from FOOD has been valuable for many discussions within the EPOS projects. FOOD has been developed as the result of two Masters theses only, and thus no further work on implementation may be expected. However, FOOD is, and may continue to be, used as a testbed for the other database work in our group. We have a framework for testing out new ideas, how they apply and fit in. This is very useful, and much of the experience and ideas as gained within context of FOOD, will be brought into other national and international database projects. This applies to modeling, prototyping and general ideas and competence.

Within the database work for REBOOT[10], we have made considerable use of the results from

[10] REBOOT - REuse Based on OO Techniques, is an ESPRIT 2 project working on providing an environment for Reuse of software artifacts, and the use of OO methods is central in this project. It lasts from 1990-1994, and

FOOD. The REBOOT data model is just structurally OO, but many of the ideas from FOOD has proven very valuable within the reuse framework.

We think the major importance of our work is the development of a complete model capable of solving most of the problems as experienced within the application areas addressed. Also, we have for many parts of the data model gone much further than what is usual among most other OODBs, to some extent satisfying almost all requirements for extended database functionality as posed in [Sto90]. With respect to this, FOOD may be regarded as an attempt to close the gap between the two manifests - a gap we feel is not necessarily present at all.

The field of OODBs is very interesting with respect to the future. We envisage an increasingly growing interest from various user communities for considering the new technology. The OOPSLA '90 [Mei90] conference had eight OODB vendors demonstrating their products, some of them with a quite large number of installations. Also, a separate panel session was dedicated to the field, the panelists concluding with a large number of issues for further research, as for instance: Common framework, concurrency issues, query languages, relationships, support for making schema modifications, transaction models and constraints. That is, these are issues which to a large extent have been addressed within FOOD.

Also, we foresee the two manifests will generate a fruitful discussion of future database functionality, the two camps approaching each other, finding the differences are not very big, really: There is a common agreement that new database technology is required.

# References


[ABD+89] Malcolm Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The Object-Oriented Database System Manifesto. In *Proceedings from DOOD '89, Kyoto, Japan*, December 1989.

[BH89] Anders Bjørnerstedt and Christer Hültén. Version Control in an Object-Oriented Architecture. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 18, pages 451–485. Addison Wesley, September 1989.

[BKKK87] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proceedings of SIGMOD*, pages 311–322, May 1987.

[Bra90] Svein Erik Bratsberg. FOOD - Supporting Explicit Relations in a Fully Object-Oriented Database. In *Proceedings from IFIP 4th TC2 Conference on Database Semantics; Object-Oriented Databases: Analysis, Design and Construction. Windermere, UK*, July 1990.

[Dit88] K.R. Dittrich, editor. *Proceedings of the Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems*, September 1988. Lecture Notes in Computer Science, no. 334.

[Hol88] Per Holager. Elements of the design of a change oriented configuration management tool. Technical report, ELAB. SINTEF, Trondheim, Norway, February 1988. Technical Report STF44 - A88023.

[LCD+89] Anund Lie, Reidar Conradi, Tor M. Didriksen, Even-André Karlsson, Svein O. Hallsteinsen, and Per Holager. Change Oriented Versioning in a Software Engineering Database. In *Software Configuration Management Workshop, Princeton, New Jersey, USA*, October 1989.

[LH90] Barbare Staudt Lerner and A. Nico Habermann. Beyond Schema Evolution to Database Reorganization. In *Proceedings of the Joint Conference on Object-Oriented Systems, Languages and Applications (OOPSLA) and ECOOP, Ottawa, Canada*, pages 67–76, October 1990.



includes 124 manyears in total. The Norwegian efforts in this project are supported by NTNF - The Royal Norwegian Council for Scientific and Industrial Research.

[LHC90] Anund Lie, Per Holager, and Reidar Conradi. Change-oriented versioning: Rationale and evaluation. Technical Report 16/90, EPOS report 101, 11 p., Division of Computer Systems and Telematics, Norwegian Institute of Technology, April 1990. Accepted at 3rd Int'l Workshop on SW Engineering and its Applications, Toulouse, France, 3-7 Dec. 1990.
Accepted at NIK'90, 27-28 Nov. 1990, Bergen.

[Lie90] Anund Lie. *Versioning in Software Engineering Databases.* PhD thesis, Division of Computer Systems and Telematics, Norwegian Institute of Technology, January 1990.

[Mei90] Norman Meirowitz, editor. *Proceedings of the Joint Conference on Object-Oriented Systems, Languages and Applications (OOPSLA) and ECOOP, Ottawa, Canada*, October 1990.

[Mun90] Bjørn Munch. Change-oriented versioning. Master's thesis, Division of Computer Systems and Telematics, Norwegian Institute of Technology, June 1990. EPOS report no. 110.

[OB90] Erik Odberg and Svein Erik Bratsberg. FOOD Object-Oriented Database. In *Proceedings, Norsk Informatikk Konferense 1990, Bergen, Norway*, November 1990.

[PS87] D. Jason Penney and Jacob Stein. Class Modification in the GemStone Object-Oriented DBMS. In *Proceedings of the conference on Object-Oriented Systems, Languages and Applications (OOPSLA), Orlando, Florida, USA*, pages 111–117, October 1987.

[Sto90] Michael Stonebraker. Third-generation database system manifesto. In *IFIP TC-2 Working Conference on Database Semantics in Windermere, (Lake District) U.K., 2-6 July, 1990*, July 1990. Also published in SIGMOD Record.

[SZ86] Andrea H. Skarra and Stanley B. Zdonik. The Management of Changing Types in an Object-Oriented Database. In *Proceedings of the conference on Object-Oriented Systems, Languages and Applications (OOPSLA), Portland, Oregon, USA*, pages 483–495, September 1986.

[Zdo86] Stanley B. Zdonik. Maintaining consistency in a database with changing types. Technical report, Brown Univeristy, Department of Computer Science, 1986.