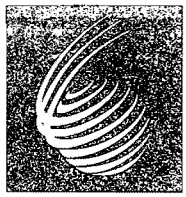

CLRC

Technical Report
RAL-TR-95-014

Comparison of ONTOS and the Manifesto for Object-Oriented Database Systems

C Rolker

May 1995

COUNCIL FOR THE CENTRAL LABORATORY OF THE RESEARCH COUNCILS

Enquiries about copyright, reproduction and requests for additional copies of this report should be addressed to:

The Central Laboratory for the Research Councils
Library and Information Services
Rutherford Appleton Laboratory
Chilton
Didcot
Oxfordshire
OX11 0QX
Tel: 01235 445384 Fax: 01235 446403
E-mail library@rl.ac.uk

ISSN 1358-6254

Studienarbeit:

Comparison of ONTOS and the Manifesto* for Object-Oriented Database Systems

by

Claudia Rolker

Advanced Database Section
Data Engineering Group
Systems Engineering Division
Computing and Information Systems Department
Rutherford Appleton Laboratory,
Chilton, Didcot
OXON OX11 0QX
UK

Universität Karlsruhe
Postfach 6980
76128 Karlsruhe
Germany
email: s_rolker@ira.uka.de

Rutherford Appleton Laboratory, February 1995

---

* Atkinson, M.; Bancilhon, F.; DeWitt, D.; Dittrich, K.; Maier, D.; Zdonik, S.: The object-oriented database system manifesto; in *Deductive and Object-Oriented Databases. Proceedings of the First International Conference (DOOD89)*, p. 223-240;

Acknowledgments

This report is the result of 4 months intensive work between November 1994 and February 1995. The work took place at Rutherford Appleton Laboratory (RAL) which is part of the Council for Central Laboratory of the Research Councils, UK.

I am a student at the Technical University of Karlsruhe, Germany (TUK) and I was allowed to do my "Studienarbeit" as a student project at RAL.

I thank Prof. P. Lockemann (TUK and FZI), Prof. K. Jeffery (RAL) and Eric Thomas (RAL) who enabled me to do this project. I thank my supervisor John Kalmus (RAL) for his scientific guidance during my whole project, for his corrections of this report and his administrative work.

Thank you as well to Hartmut Schreiber (FZI, Germany) for his scientific support and administrative work.

Last, but not least, I would like to thank all those from RAL and TUK who have discussed particular scientific problems which arose during the project: Una l'Estrange (RAL), Damian Mac Randal (RAL), Dr. Michael Wolverton (RAL), Dr. Gerd Hillebrand (RAL), Manfred Maennle (TUK), Harald Weidner (TUK), Gerhard Wickler (RAL), Myles Chippendale (RAL), Dr. Simon Dobson (RAL), Dr. Chris Wadsworth (RAL).

Finally, I thank all colleques at RAL who created a relaxed atmosphere and who contributed to a pleasant stay for me in Didcot, which I enjoyed very much.

Contents

1 Introduction

ONTOS is one of the currently available commercial object-oriented database systems (OODBSs). It was developed by Ontologic Inc. and has been available since 1989. Since that time there have been several versions of ONTOS. We use ONTOS Version 2.2 in our evaluations. Every version usually has significant new code or new functions and an improvement in the functionality or speed of the product. ONTOS is one of the systems that is widespread and seems to fulfil the user's idea of an OODBS. But is this the same idea which researchers have of an OODBS? The basic aim of the project is to answer this question.

We have based our investigation on the paper "Object-Oriented Database System Manifesto" by Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier and Stanley Zdonik, written in August 1989 [ABDB 89]. It describes the main features and characteristics that a system must have to be considered to be an object-oriented database system.

The report is organized as follows. Section 2 gives an overview of the OODBS Manifesto. Section 3 explains the ONTOS architecture and how the user's application interacts with tools and utilities.

Some of the features demanded by the Manifesto can be checked by writing an application. Others are internal, so we have to rely on what the ONTOS user manual says. Our application is a management of a video-shop which is described in Section 4 in more detail. In Section 5 and 6 we compare the features demanded by the Manifesto with the actual features of ONTOS and give examples, where possible, using our video-shop application.

Section 7 summarizes all examined features of ONTOS and assesses ONTOS with regard to the theoretical demands of OODBS.

2 The Object-Oriented Database System Manifesto

One of the major motivation for the object-oriented data model was a desire to bring some of the concepts of object-oriented programming languages (such as Smalltalk and C++) into database systems. The large number of concepts that have been imported into object-oriented research means that there is really no common object-oriented data model. Different OODBS products offer different capabilities from the possible set. This is detrimental to product sales, since commercial business prefers to deal with a standard interface.

Because of the lack of a common data model, the lack of formal foundations and strong experimental activities in the OODBS field, six highly respected practitioners in the field wrote a paper entitled "The Object-Oriented Database System Manifesto" in 1989 [ABDB 89]. They attempted to prioritize features of OODBS. They distinguished between mandatory, optional and open features.

The 13 *mandatory* features, also called "Golden Rules" for an OODBS, are those ones which a system *must* satisfy in order to be termed an object-oriented database system. An OODBS is a database management system and an object-oriented system. Along these lines the mandatory features can be subdivided into

- DBMS features:
  persistence, secondary storage management, concurrency, recovery and an ad hoc query facility

and

- features of object-orientation as usual in object-oriented programming languages:
  complex objects, object identity, encapsulation, types or classes, inheritance, overriding combined with late binding, extensibility and computational completeness.

The 5 *optional* features are the ones that are desirable because they make the system better, but which are not mandatory. Most of them are for support of special applications (CAD, CAM, etc.). The so-called goodies are: multiple inheritance, type checking and type inferencing, distribution, design transactions and versions.

Last, the *open* features which are those where the designer of the OODBS can select from a number of equally acceptable solutions. The Manifesto leaves these features as open choices. The authors see no point in preferring one alternative to another. The open features are: programming paradigm, representation system, type system, uniformity.

We examined ONTOS only in regard to the mandatory and the optional features.

The authors of the Manifesto describe the characteristics a system must have to be an object-oriented database system, but they clearly see this paper not as the definition of OODBS for all eternity but as a first step to characterize OODBSs.

3 ONTOS Overview

3.1 Process and netarchitecture

ONTOS is the successor to Vbase and is a distributed object-oriented database system that uses client/server architecture to distribute the database around a network of homogeneous workstations. It runs on UNIX environments such as SUN, HP Apollo and DEC workstations. ONTOS is also available for the OS/2 operating system. Its approach toward object-orientation is to add persistent storage to the C++ programming language.

A client is created by linking the ONTOS Client Library into a C++ application. The client communicates with the "primary server" which is the primary manager of the distributed database.

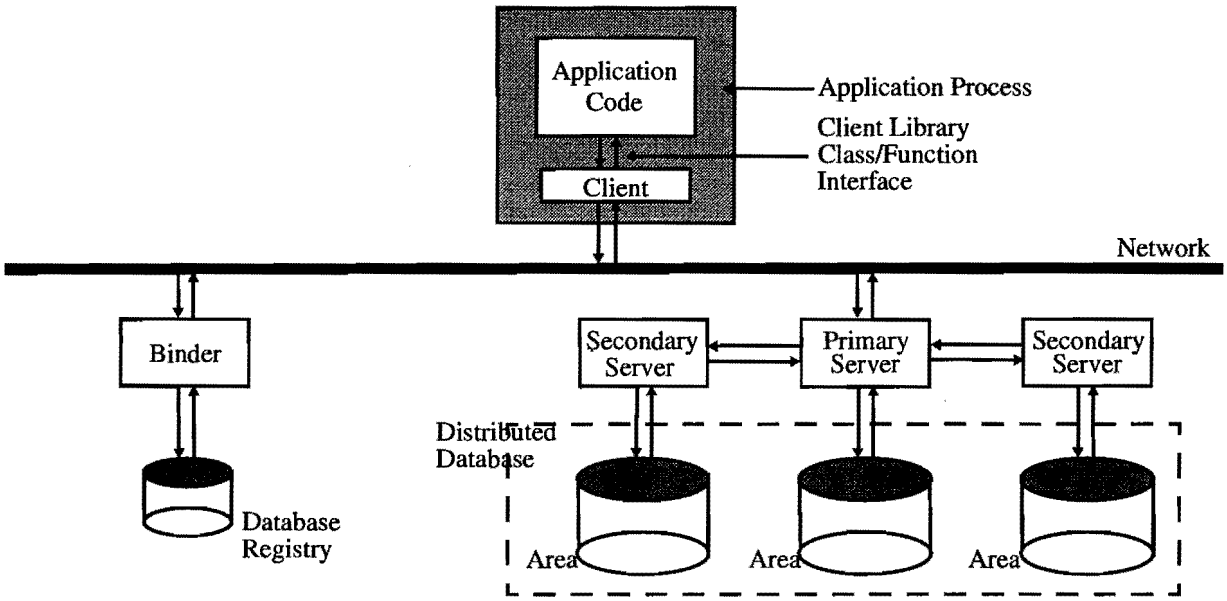

Figure 1 -The client- server architecture

The database is divided into areas. Each area has a dedicated server process, but any I/O on an object in any area of the database is channelled through the primary server to the server of the appropriate secondary area. The primary server handles storage and retrieval (get/puts) of objects in its own area in addition to managing the secondary servers.

Each of the areas is a physical file where the objects are stored. One of the areas must include the kernel area, which consists of objects in the ONTOS-Schema (= ONTOS DB metaschema classes, which the installation of ONTOS DB is shipped with).

The user must specify one of the areas as the primary area, whereas the other areas become secondary areas. Along these lines the user defines implicitly the primary server process. The usual policy is to make the primary area the one that experiences the most get/puts from client applications since it is the default storage for new objects and since it is the only area in the database that has a direct server connection to clients.

The binder acts as a global network service and sits out on the network. It is responsible for connecting clients to servers managing the database of interest. The binder consults the database registry for information. The registry is the control structure that stores the list of registered areas, hosts, databases and their respective mappings. The binder is the only component which has direct access to the database registry. So if the binder breaks down, no database is accessible no more and the whole system stops.

The user can only access the registry by means of the DBATool, on behalf of which the binder acts to the registry as a server. The DBATool helps the user to configure the database and to look at the already registered databases and areas.

Each server has a cache, which is a chunk of memory and allocated at the time that an area is opened. The cache is used to hold segments of data that have been read from or are to be written to the database. A segment in ONTOS is the unit of transfer from disk into the server cache, which contains a group of objects. All objects in a segment are transferred into the cache when any object in that segment is activated. The default cache size is set low, to 1 Mb, in order to conserve memory. The maximum size is 2 Mb.

Objects are requested singularly or in groups by the client, are retrieved by the servers and handed to the client. These objects are activated in the client application's virtual memory and manipulated as C++ structures. When the application finishes with these objects, it deactivates them, optionally deallocates their memory, and passes them to the client. The client portion transfers them back to the servers where they are kept in the server cache until it makes sense to make the changes to the area.

3.2 Steps to run an ONTOS DB application

In the following the steps to run an ONTOS application are described:

1. The user configures the database with the help of the *DBATool* and registers it so that it is accessible by client applications.
2. The user uses the ONTOS classify utility which processes his C++ header files and which builds a representation of this information in the database. The classify utility can build the representation of C++ classes, including data members and member functions, as well as free functions. These representations allow ONTOS DB to store data whose structure is defined by a C++ class, and to provide the user access to this data, its schema, and to any free functions represented in the database.
   When the user submits a C++ source file, containing one or more persistent class definition, the classify utility reads the file and creates the objects that correspond to each class description and puts theses objects into the database. These objects are:
   - A *Type* object
     The *Type* object contains the specifications that are used to construct instances of the class in the database. They are used by the database as templates for handling objects (instances) of the Type and for converting object references between their in-memory and database forms. *Type* objects are also available to the application as runtime-accessible descriptions of persistent classes.
     When a *Type* is generated, *Procedure* and *PropertyType* objects that correspond to the class members are also generated and put into the database.

- *Procedure* objects
  Procedure objects represent methods and their signatures.
- *PropertyType* objects
  These are run-time accessible representations of class members.

These created objects are instances of the ONTOS DB *metaschema classes*. The user can use control files with the classify utility ie. to break the created schema up into parts that can be stored in different areas of the database.

3. The user compiles his C++ program modules with the cplus compilation utility which provides a transparent way to compile extra information required by ONTOS DB into C++ program modules and which links the modules with the ONTOS DB client library.
4. The user executes his application. When the client application calls OC_open to open the database, the binder looks up the mapping between the database name and its areas, starts the server (the database process) for the primary area (secondary servers are not started until access to their specific areas is required), and then returns a handle that the client uses to connect to the server. If a client calls OC_open on a database and that database has an area with an already active server, the binder connects the new client to the existing server. The binder's job is finished once the primary server is activated for the database. The server handles transaction starts, commits and get/puts to the database.

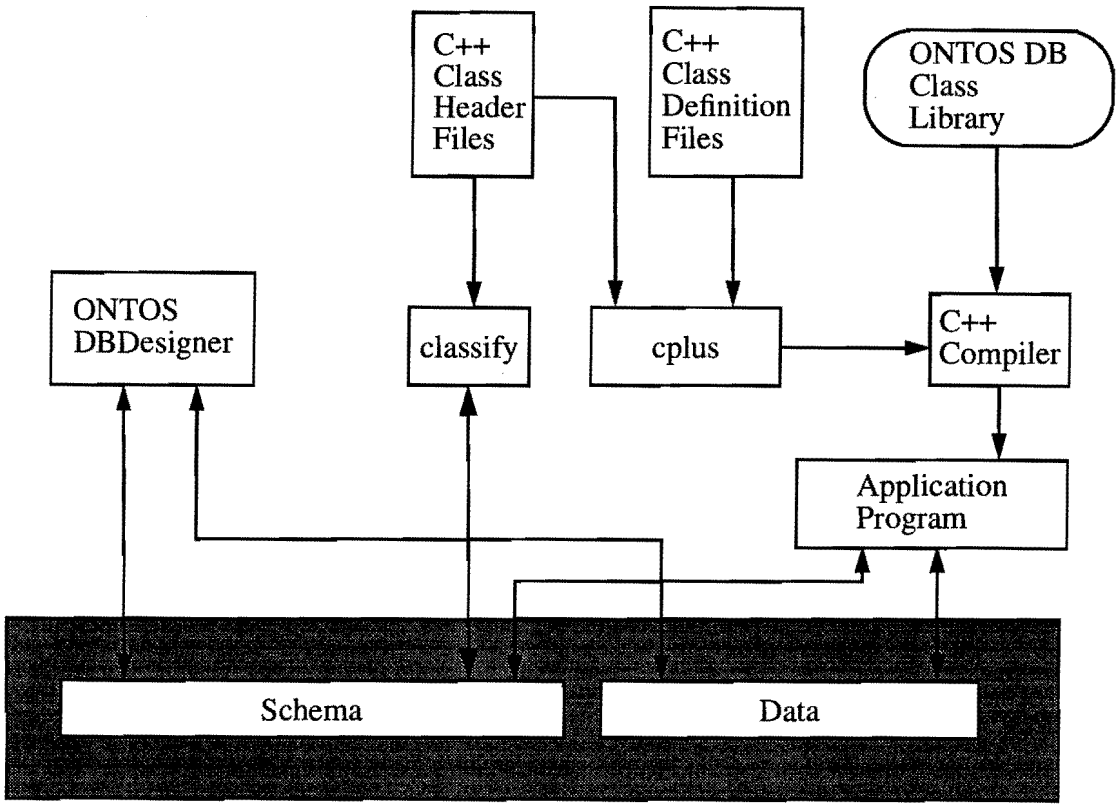

Figure 2 -ONTOS tools and utilities

The user can also work with the DBDesigner, which is an X windows based tool used for schema design and browsing. It shows the user a schematic diagram of the classes in a selected database. It lets him view the schema and contents of the database and allows him to modify them by adding, deleting and rearranging the elements of the diagram. It also provides automatic generation of C++ header files to implement the designs the user creates with this tool.

4 Example

4.1 Description of example

The goal of the example is to check as many features, commands and characteristics described in the ONTOS manual as possible in order to be sure that they are really provided by the system while retaining a fairly simple, "common sense" example which is easy to understand. Moreover, we wanted to get the feeling for this OODBS. The application chosen is a videoshop management system.

Our videoshop software deals with adding new videos to the stock, getting information about the videos (titles, number of copies, medium: tape or LDLaserDisk) and removing old, worn out videos. Before someone can rent or reserve a video for his first time, he has to give his address and further personal information. Our software generates a unique membership number for him, which the member must use whenever he wants to rent or reserve a video. Our software manages both rental and reservation of videos. Our imaginative videoshop offers a special service: it picks up and delivers videos which of course entails additional costs. Members may pay by cash, by creditcard or use a special account at our videoshop, which adds together all costs during a month, which are then paid by cash.

The main task of the software is of course to manage who has reserved/rented which video and when.

4.2 Object-role-modelling for the example

Before we implemented the videoshop example we made an object-oriented design of our application to ensure clarity. We took the object-role modelling approach from T.A. Halpin, University of Queensland, Australia [Halpin 94]. It is very similar to entity-relationship modelling, but has some additional features. We chose this design approach because it is an up-to-date approach which has recently attracted much interest in UK academic communities. The technique was new to the author, but now she has gained some experience in it.

In the object-role modelling technique entity types (object types) are depicted as named ellipses. Predicates are shown as named sequences of one or more role boxes, with the predicate name starting in the first role box. Each role is ordered, from its first role box to the other end. A role is mandatory for an object type if and only if every object of that type which is referenced in the database must be known to play that role. This is explicitly shown by means of a mandatory role dot where the role connects with the object type. If two or more roles are connected to the same mandatory role dot, this means the disjunction of the roles is mandatory (each object of this type must play at least one of these roles). Exclusion is indicated by the symbol ⊗ that is one role or another, but not both. A bar across n roles of a fact type indicates that each corresponding n-tuple in the associated fact table is unique (no duplicates are allowed for that column combination). Finally, subtypes are defined as directed line segments from subtypes to supertypes.

Figure 3 shows the result of the object-role modelling for our video-shop software. A video is a subtype of a videomedium, which is the pure tape or LDLaserDisk, and a subtype of a videomovie, which is defined by a title, a duration, a main star and the kind of movie. The video has several copies which belong to the video-shop. This is the "video-part".

The "member-part" at the bottom of the diagram consists of the members that are identified by their member number. Each member is a person which has a name and an address.

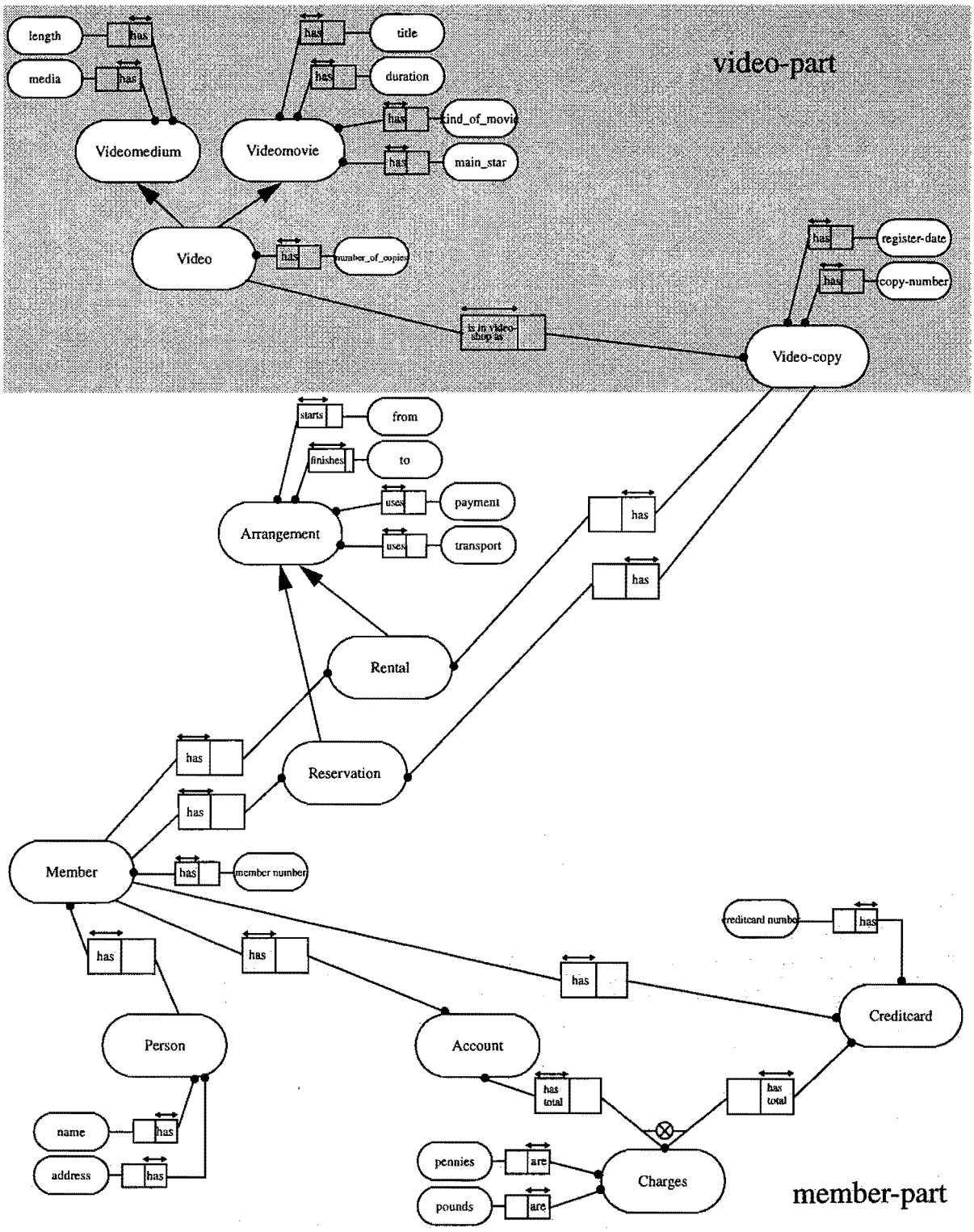

Figure 3 -Object-Role Model of the video-shop example

If a member always pays by cash, then he does not need to have an account or a saved creditcard number. If a member has an account, then it is unique. He can share this account with other people if he wishes to (e.g. relatives). The same is true for the saved creditcard number. To each account or saved creditcard number a total is associated which is initialized to zero when the objects are created.

The "member-part" and the "video-part" are connected through rental and reservation types which are subtypes of arrangements where the rental/reservation conditions are defined. The conditions are for which period of time the member wants to rent/reserve the video, which kind of payment he will use, and if he want the video-shop to pick-up or deliver the video from/to home.

4.3 Description of the implementation

The class video has two superclasses: video-medium and video-movie. Along these lines we experimented with multiple inheritance. The video instances are organized in two directories, depending on their medium (tape or LDLaserDisk). The reason for this is that a member who asks for a special video probably only has one video system at home and so the search can be restricted to one dictionary from the beginning. This reduces the search time.

Each video instance itself comprises a dictionary which maps the copy number to the actual video-copy. When an old video-copy instance is removed from the stock, we only have to delete the entry in our dictionary. When a video-copy is added to the stock, we only have to go along in our dictionary and find the first copy-number without a mapping object. This trick is used to implement the 1:n relationship.

Each member can rent/reserve a number of videos. That is why we used here the same principle, but this time we made use of sets instead of dictionaries to replace the 1:n relationship. The instance member has references to an account instance or creditcard instance, if necessary. Within those two an instance of charge is embedded.

The rental and reservation classes are subclasses of the arrangement class. Along these lines we checked simple inheritance. All the big ellipses in Figure 3 are transferred into classes. All the smaller ellipses in Figure 3 are implemented as attributes to the next ellipse transferred into a class.

All role objects are transferred into references from one instance of a class to an instance of another class, apart from those exceptions described above.

Throughout the implementation of the rental class and the reservation class as two distinct classes we have to be very careful in our application program, when a reservation becomes a rent. This action takes place within a transaction so that in case of a crash no inconsistent state remains. The deletion of the reservation object and the creation of the new rental object must be in one transaction.

The implementation of the example was actually not very hard, but the error messages which are sent out either at compile time or at runtime were not very helpful. Most run-time errors could only be found by using a debugger or adding output messages into the program code.

Moreover, we could not manage to achieve a bidirectional reference although the manual says that it is possible.

5 Comparison of Mandatory Features

5.1 Persistence

5.1.1 The Manifesto's demands

Persistence means that an object can persist beyond application session boundaries. The object can be retrieved by another application session and will have the same state and relationships to other objects as at the time when it was saved.

The Manifesto demands that objects of any type, including arbitrary complex user-defined types may be allocated in either persistent or volatile store. The user should not have to explicitly move or copy data to make it persistent.

5.1.2 Persistence in ONTOS

In ONTOS, persistence is orthogonal, since it is independent of its type, whether or not the object is persistent.

A separation of classes for persistence and volatility is made at the level of the class description. The persistent class must be derived from a superclass, the class Object. A persistent class description must satisfy the following criteria [Ontos 92a]:

1. It must derive from the class Object
2. A special constructor member function (in addition to the usual C++ constructor) must be included. It is used to search for an object in the database and move it to an application programm. It is the activation constructor and takes an argument typed as "APL*".
3. A *getDirectType()* member function must be included to return a pointer to the persistent representation of the class in order to access it from a program.
4. If a class description has a destructor, then a Destroy function must be added to run when any exceptions are raised to remove the object from memory.
5. If a class description contains an operator *new*, then its signature must be identical to that which is used by ONTOS to allocate memory for a newly allocated object.

When an instance of a persistent class is created, it is not stored in the database until the putObject() function is called, which takes care of putting the instance to the database and the user does not have to deal with this problem. The putObject() function belongs to the class Object. So if the programmer creates an object of a persistent class, but does not use the putObject() function, this object is volatile. On the other hand, when a persistent object is retrieved from the database and is changed in some data members, the changes are not updated in the database if the putObject() function is not called and so the changes can be made explicitly persistent object in the database.

If an instance of a volatile class is created, this cannot be made persistent.

5.2 Complex objects

5.2.1 The Manifesto's demand

Complex objects are created by applying constructors to objects, building arrays or sets of objects, and naming the resulting aggregate as a new object. So complex objects are objects which have a complex structure consisting of other sub-objects. The structure can be (for example) a tuple, a set, a bag, a list or an array. But the Manifesto demands at least set, list and tuple. The sub-objects can be simple objects like integers, characters, byte strings of any length, booleans and floats (there might be more!) or again a complex object.

Moreover the Manifesto demands that the object constructors (tuple, set, etc.) must be orthogonal which means that it is possible to construct every complex object that you can imagine when you know the simple objects and the constructors the OODBS offers. You should not have to take into account any prohibitions, as for example in the relational model where a set can only be constructed with tuples as members.

Having complex objects the OODBS must provide some operators to retrieve, delete, make a deep copy of a complex object. (In a deep copy the whole complex object is copied inclusive of referenced objects, whereas in a shallow copy only the first layer is copied but not for example copies of referenced objects are created.) Apart from this, other operations may be defined by the user of the system. Nevertheless two types of references must be provided:

- is-part-of reference: which has the consequence that with deletion of the comprising object the referenced object is deleted
- general reference: which has the consequence that a deletion of the comprising object does not affect the referenced object at all

5.2.2 Objects in ONTOS

5.2.2.1 Simple objects in ONTOS

Complex objects in ONTOS are based on the simple objects char, int, char*, short, long, float, double, which C++ offers, and on OC_Boolean which is defined as a enum type:

enum OC_Boolean {FALSE=0,TRUE=1}.

So all demanded simple objects of the Manifesto are offered by ONTOS.

5.2.2.2 Structures of complex objects in ONTOS

One of the demand structures of an object is the tuple structure which is implicitly given by the way new classes are defined in ONTOS (is defined as in C++). In our video-shop example we defined the class "member" which has a tuple structure and stores information about the members of the video-shop. Each "member" object consists of

- an integer number of the member
- an object of the class Person (which stores information about the person's name, address etc. and we implemented it as a pointer to the object)
- an object of the class Account (which stores information about a video account and the total and we implemented it again as a pointer to the object)

- an object of the class Creditcard (which stores a member creditcard number and the total and which we implemented again as a pointer to the object)
- a set ReservedVideoSet (which stores all videos that are reserved by that person, the dates, how the member will pay later for renting the videos. We implemented it again as a pointer to the object)
- the set RentedVideoSet (which stores all videos that are rented by that person, the dates, how the member will pay later for renting the videos. We implemented it again as a pointer to the object)

The class "member" is declared as follows in ONTOS:

```
class member: public Object
{
        int member_number;
        Reference has_creditcard;
        Reference has_account;
        Reference is_person;
        Reference reserves;
        Reference rents;

public:
        member();
        ..... // more constructors and methods of this class
}
```

As we have used abstract pointers (declared with "Reference") you cannot see which object type is referenced. But it is guaranteed with the help of the methods of the member class.

Apart from tuples, ONTOS offers some other structures. They are represented by the persistent class aggregate and its derived classes: set, list and association. The last one itself has subclasses array and dictionary. The aggregate class defines some common properties for all subclasses: 1) memberSpec, which return a type of the aggregate, and 2) cardinality, returning the number of objects. Aggregates also specify a number of procedures: isMember, isSubSet, checkMemberSpec, getIterator, getClusterSize, putCluster. Each of the aggregates defines an isSimilar procedure.

Lists are ordered, unkeyed aggregates that represent linked lists, sequences, queues or stacks. A list stores members serially; each member has a position in the list. Insertion into the List at a particular position increments the position of all members following that position. Removal of a member does the opposite. Lists are implemented so that insertions and removals are fast relative to arrays and ordered dictionaries. Insert, remove, and access operations near the start of the list or at the very end are faster than those in the middle [Ontos 92c].

Sets are unbounded, unordered aggregates. Set members can be inserted, removed and tested for membership. Unlike the other aggregate-based classes, sets do not support multiple entries for the same element [Ontos 92c].

In an *array* keys must be the continuous range of integers between the specified lower and upper bounds, either of which may be positive, negative or zero. All the elements of an array are allocated and initialized to NULL. Arrays can be resized by specifying new bounds [Ontos 92c].

Dictionaries map tags of any class to elements of any other class. Unlike an array's indices, a dictionary's tags can span a wide range of values without incurring overhead for "unused" tag values that fall within the range but have no entries. Dictionaries may be ordered or unordered. Ordered dictionaries use B*tree access structures to sort entries based on the relative ordering of their tag values; unordered dictionaries use hash-table access structures. Dictionaries may be defined to allow for duplication (that is a dictionary may have two different objects that have the same tag) [Ontos 92c].

ONTOS fulfils the minimal set of constructors that the Manifesto demands.

In ONTOS all constructors can be applied to any object and are orthogonal. So also this Manifesto demand is fulfilled.

5.2.2.3 Operations on complex objects in ONTOS

There are two major possibilities of how objects can belong to a tuple: either referenced or embedded.

In a retrieval, embedded objects are always retrieved with the comprising object and deleted if the comprising object is deleted.

ONTOS offers two kinds of references: abstract reference which provides the user with some methods or direct reference which is the kind of reference used in C++. Using abstract references the referenced object is automatically retrieved by traversing the reference if the referenced object has not been in memory before. Whereas when you use direct references it is left to the user to retrieve the referenced object before traversing the reference. As a difference to the Manifesto, both kinds of references leave the deletion of the referenced object with the comprising object to the user, but it is quite easy to implement.

In our video-shop example we have embedded the object total of the class charge into the class creditcard. So with the creation of a creditcard object the object charge is created and initialized to zero and with the deletion of a creditcard object the total object is deleted:

```
class creditcard: public Object
{
private:
        char *creditcard_number;
        charge total;

public:
        creditcard(char * the_number);
        creditcard(APL* theAPL);

        Type *getDirectType();

        char* get_number();

        void get_charge();
        void set_charge(int the_pounds,int the_pennies);
        void add_charge(int the_pounds,int the_pennies);
        OC_Boolean isZero();
        void dump();
};
```

In the class member we want to delete the according personal information with the deletion of the member. As the usual deleteObject() method does not do this, we had to redefine this method. But we do not necessarily want to delete the creditcard information with the deletion of a

member because it may belong as well belong to another person for example to the spouse, to the parents etc. So we do not add this in the redefined method deleteObject() of the class member.

```
void member::putObject(OC_Boolean deallocate)
{
        ((Object*)isPerson.Binding(this))->putObject(FALSE);
        Object::putObject(deallocate);
}
```

The retrieval of the objects of aggregates is done with the retrieval of the aggregate, but again the deletion of the whole aggregate is not supported by ONTOS, but can easily be arranged by the user by a redefinition of the deleteObject() method.

A method that makes a deep copy of a complex object is not provided by ONTOS, but can be defined by the user. Depending on how complex the object is the definition might be not that easy.

5.3 Object identity

5.3.1 The Manifesto's demands

Every database system must have some way of distinguishing one object from another. This can be done in a value-based system by introducing explicit object identifiers, but then the user has to insure the uniqueness of object identifiers. But this is not a very neat solution. That is why the Manifesto demands the system itself to define and maintain unique identifiers for objects. This implies that objects have existence independent of their values. So even if they have the same values they can coexist.

This results in two definitions of equivalence:

- two objects are *identical* when they are the same object
- two objects are *equal* when they have the same values

So in an identity-based model two objects can share an object and when in one of the objects the shared object gets new values, these can be seen as well in the other object. For example object1 and object2 comprise object3, then when in object1 object3 is changed, it is automatically changed in object2 as well because it is the same object.

Moreover, the Manifesto demands operations like object assignment, object copy (deep and shallow) and tests for object identity and object equality (deep and shallow). One object is assigned to another when both objects are already created and memory is already allocated and now the values are copied. We make a copy of an object when we create a new object and initialize it with the other object's values.

5.3.2 Identifying objects by name

In ONTOS the user may assign a name to an object, so that there is a named access to the object in the database. The name is treated as a kind of persistent variable. Names can serve as an entry point into the database for activating the first few objects. From there, the application can access further objects either by name or by object-to-object references. The object bound to the name can be changed. To find an object by a name ONTOS uses directories which map names

to objects. Within one directory the names must be unique. But there can exist several objects with the same name when their mappings are declared in different directories. The database name space is organized into a hierarchy of directories. So within the search for an object the directories where to look for the object must be given to let the system know in which order the directories should be looked for the object.

It is not mandatory to give an object a name. This would be too much effort for the user because he has to find unique names. That is why ONTOS manages and maintains objects as well with an interior object identifier for every object.

5.3.3 Object identity provided by the system

The ONTOS database is a single uniform object storage space. Within this storage space each object has its own unique identity (UID), which is persistent and immutable. The identity comes into being when the object is created and continues to represent that object from then on. The UID is a unique 64-bit value [Ontos 92a], not exported to the user application. Thus the number of objects is bounded (but is very large!).

Other objects in the database can use the UID to unambiguously reference the object. We used this feature in our video-shop example as it is possible that two members want to pay with the same creditcard. Proceeding on the assumption that the husband is already a member of the video-shop, but his wife wants to become a member as well, it sounds reasonable to debit their costs to the same account at the video-shop. This is implemented as follows:

(In the comment we play through the above described husband-wife situation.)

```
.....
char mem_obj_name[40];                          // is the object name of the object wife
concatenate(member_nr,"mem",mem_obj_name);

member *new_member = (member*) OC_lookup(mem_obj_name);// new_member is the object wife
account *new_account;                           // new_account will be the wife's account
int other_nr;                                   //other_nr is the husband's member number

char other_mem_obj[40];                         // will be the object name of the object husband

if (!new_member == NULL)                        // test if wife exits as an object in our db
{
      char answer;
      cout << "Use existing account? (y=yes n=no)\n";
      cin >> answer;
      switch (answer)
      {
            case 'n':                           // case 1: wife wants to have her own account
            case 'N':
            new_account = new account();        // create and initialize the wife's account
            new_member->set_account(new_account);// connect object wife with wife's account
            new_account->putObject();           // save change in db
            new_member->putObject();            // save change in db
            break;

            case 'y':                     // case2: wife want to use her husband's account
            case 'Y':
            cout <<"from which member\n";       // gives the husband's member number
            cin >> other_nr;
            concatenate(other_nr,"mem",other_mem_obj);
            member *other_member = (member*) OC_lookup(other_mem_obj);
                                                // search for object husband in db
             if(other_member ==NULL)            // test if husband exists as an object in our db
```

```
            {
                cout <<"does not exist";
                return;
            }
            new_account = other_member->get_account();          // find husband's account in db
            new_member->set_account(new_account); // connect husband'saccount to object wife
            new_account->putObject();                  // store changes in db
            new_member->putObject();                   // store changes in db
            break;

            default:
            cout   << "Sorry, "
                   << answer
                   << " is an illegal choice. Try again!\n";
        }

    }
    else cout<<"member doesn't exist! \n";
    .....
```

The Reference is hidden behind the get_account() and the set_account() member functions of the class member, which are defined as follows:

```
account *member::get_account()
{
    return (account *)has_account.Binding(this);
}
```

The *Binding* function returns a pointer to the referenced object and activates the object if necessary.

```
void member::set_account(account* the_account)
{
    has_account.Reset(the_account,this);
}
```

The *Reset* function is used to make a Reference to a new object account called the_account.

So after this procedure two member objects (of the video-shop) share an object account, and every change in this object can be found out either through the first member or through the second member.

5.3.4 Operations in context with object identity

To compare two objects for identity ONTOS provides a member function of the class Entity which is the base class for all objects:

```
virtual OC_Boolean operator==(Entity& anotherEntity)
```

In case of simple objects, this function compares their values.

But ONTOS does not give much help for comparing two complex objects for equality. If these are objects of the list class, dictionary class, array class or set class, there is a member function called isSimilar.

In the case of dictionaries these functions compare two dictionaries and return TRUE if all the attributes of the two dictionaries are the same. They must contain the same (identical) objects and must have the same memberSpec which means that in the declaration of the dictionaries the same class from which the members of the dictionaries must come must be specified. Moreover, the two compared dictionaries must have the same values for isOrdered and hasDuplicates to return TRUE as the result of the comparison.

In case of a list comparison this function returns TRUE if both lists contain the identical members (and no additional members) in the same order.

In case of a set comparison this function returns TRUE if both sets have exactly the same members.

In case of an array comparison this functions returns TRUE if both arrays contain the same members at the same index and have identical values for LowerBound and UpperBound.

But ONTOS provides no function for deep equality. It must be implemented by the user, which can be hard depending on the complexity of the objects.

In ONTOS an object is assigned to another by the assignment operator which has the form:

X& X::X operator=(const X&);

There are a number of problems when the predefined operator is used. The assignment fails for example for persistent classes that have a Reference object as one of their data members. That is why the assignment operator has to be redefined for every class to ensure that it works.

The same applies to copies of objects. The predefined copy constructor causes failure for some persistent classes and especially it does not save these objects in the database. So again it should be redefined for every class. The copy constructor has the following signature:

X::X (const X&);

5.4 Encapsulation

5.4.1 The Manifesto's demands

Encapsulation is the principle that a module can only be accessed via its external interface. It strictly distinguishes between the implementation of a module, which is only visible to the implementor and therefore hidden, and its interface which describes only the way in which users can view the module. Along these lines modularity is achieved.

In an object-oriented system the unit of modularity is the object, and thus, an object is the subject of encapsulation. Looking at the object from an abstract point of view, it consists of an interface part and an implementation part.

The *interface part* is realized by a set of operations fully defining the object's behaviour. The user of the object does not need to understand how these operations are implemented or how the object is represented internally. The operations which come with an object define the only way in which an object can be accessed. This restriction holds for update and retrieval operations.

The *implementation part* of the object describes the internal representation of the object (data part) and the implementation of each of the operations (operation part). When storing an object the data and the operation names are stored in the database.

The Manifesto demands encapsulation since it allows the programmer to change the implementation of a type without any program working on that object. Implementation independence is realized. Thus, application programs are protected from implementation changes in lower levels of the system.

The Manifesto points out that in the database world the structural part of a type sometimes is part of the interface. The Manifesto admits that sometimes a system can be simplified when it allows encapsulation to be be violated under certain conditions, for example with ad-hoc queries the need for encapsulation is reduced since issues such as maintainability are not important. Thus, the Manifesto demands that encapsulation mechanism must be provided by an OODBS, but there appear to be cases where its enforcement is not appropriate.

5.4.2 Encapsulation in ONTOS

ONTOS keeps mainly to the principles of encapsulation of C++ and leaves it to the programmer of the database schema whether he wants to give the user access to the data members of not. The programmer has to specify for each data member if it is

- public : accessible by the user and within any function of the class
- private : accessible within functions of the class
- protected : accessible within functions of the class and within functions of those classes who are inherited from this class

Functions of a class have to be specified in the same way. Moreover, it is possible to give special functions of other classes access to the private elements to the class explicitly by declaring them as friend functions to the class [Lippm 92].

The ONTOS manual recommends the following practices to achieve encapsulation:

- Attributes and relationships (=pointers) are recorded as private data members.
- Member functions are supplied to provide public access to the data members.

So a "normal" declaration of a class looks like the following of the class creditcard (this is the file creditcard.h).

```
#include <ONTOS/Object.h>
#include <ONTOS/Reference.h>
#include <ONTOS/Directory.h>

#ifndef CHARGE_H
#define CHARGE_H
#include "charge.h"
#endif


class creditcard : public Object
{
private:
        char *creditcard_number;
        charge total;

public:
        creditcard(char * the_number);                          //constructor
        creditcard(APL* theAPL);                                //constructor for retrieval
        Type *getDirectType();

        char* get_number();                                     //returns creditcard_number
        void get_charge();                                      // prints out the total
        void set_charge(int the_pounds,int the_pennies);        // sets the total
        void add_charge(int the_pounds,int the_pennies);
```

```
                                        // adds to the actual total the_pounds and the_pennies
    OC_Boolean isZero();                // returns TRUE if the total = 0.0 else FALSE
    void dump();                        // prints out the total and the creditcard number
};
```

The creditcard number and the total are declared as private and therefore only the member functions can access them. Get_number() simply returns the value of creditcard_number. get_charge() prints the total and dump() prints the total and the creditcard_number. add_charge() and set_charge() manipulate the total, whereas isZero() tests the total if it is zero. These functions define the interface to the object, and are used to enforce hiding of the internal structure and state maintained by the object. The methods define ways to communicate with a creditcard object and access its data members.

The declaration of the class (as above of the class creditcard) is in header-files which are to be detected by the ".h" ending (here creditcard.h). The implementation of the methods of a class are in so called implementation-files which end with ".C" (here creditcard.C) and must include the according header-files. ONTOS demands this separation because first all header-files are "classified" with the classify-utility and the schema is built. Then the implementation files and the main-file is compiled and the application is built. With the separation of implementation and declaration it is possible to change the implementation of a member function but not the declaration and the schema.

ONTOS allows the user to get information about the structure of a class. Every class description is transformed into a *Type* object, *Property*Type objects, *Procedure* objects, *ArgSpecList* objects and *ArgSpec* objects. The *Type* specifies the class' name and its base class, its data member (*PropertyType*), its member functions and constructors (*Procedure* objects) and their arguments (*ArgSpecList* and *ArgSpec* objects). These objects are constructed automatically from C++ source code by the classify utility, which generates instances of Type from C++ class definitions. The user can find out everything what stands in the class declaration with the help of the Type objects. Moreover he can get the values of every data member independent of the declaration as public, protected or private.

SQL-queries have also access to private data members (for an example see Section 5.13).

Thus ONTOS provides an encapsulation mechanism, but if there is a need to work without it, it is possible. ONTOS appeals to the programmer to keep the rules of encapsulation.

5.5 Types and classes

5.5.1 The Manifesto's demands

The Manifesto distinguishes between two categories of object-oriented systems depending on the data structuring mechanism. It says that the database schema should consist either of a set of Classes or a set of Types. (The term 'Type' has here a different meaning than in C, Pascal, etc !)

Types emphasize the fact that all instances of the Type have the same characteristics and it summarizes the common features. The Type consists of two parts: the interface part and the implementation part. The interface part is visible to the user and consists of a list of operations together with their signatures. The implementation, which is only visible to the Type program-

mer, consists of the data part and an operation part. The data part describes the internal structure of the object's data, whereas the operation part consists of procedures, which implement the interface part.

Types should force the user to declare the structure of the objects he manipulates and the system can check that the user does not perform wrong assignments to, or manipulations on, objects. Thus Types are used at compile-time to check the correctness of the program. The Types have no special status and cannot be modified at run-time.

A Class describes the common behaviour of a set of objects. The specification is the same as that of a type, but it is more a run-time notion. A Class not only describes how an object must behave to belong to that Class. It also serves as the repository of all objects which currently belong to the class. This means that the notion of class is somewhat connected to run-time considerations. Classes are not only used for checking the correctness of a program, but rather to create and manipulate objects. Classes are first class citizens which means that the class itself is an instance of another type (metaclass). So it may be subject to operations such as creation, modification or deletion, or it may establish a relationship with another class. The metaclass provides the OODBS with a powerful tool for self-description. While providing the system with increased flexibility and uniformity, this renders compile-time checking impossible.

The Manifesto admits that these two notions are often used in both senses. It does not prescribe which approach is the best for an OODBS, but one form of data structuring mechanism should be offered by the OODBS.

5.5.2 Classes in ONTOS

We grade ONTOS as an OODBS that uses "Classes" in the sense of the Manifesto. All instances of a Class have the same structure and consist of an interface part and an implementation part.

Classes can be created in two ways in ONTOS:

The first way to run the classify utility on C++ header files that contain class definitions. This utility automatically generates instances of the metaclasses *Type*, *PropertyType* and *Procedure* from standard C++ class definitions. The classify utility processes the C++ information and uses it as model to build the corresponding *Type* object, *PropertyType* objects and *Procedure* objects within the database. During compile time all these objects exist. This way is the most common way.

The other way to create *Type*, *PropertyType* and *Procedure* objects in the database is to directly create instances of the metaclasses *Type*, *PropertyType* and *Procedure* by including the appropriate C++ code in the database application. This manner of Class creation is called *programmatic type creation* because it occurs from within the program at run-time. Along these lines, class definitions can be created, modified and deleted at run-time.

ONTOS offers the programmer of a *Type* to maintain the extent of a Class (ie. the set of objects of a given Class in the database) if he wants the *Type* to have so. After the creation of the *Type* object this is not changeable. So ONTOS makes all objects of a Class accessible to the user, if the *Type* has an extent.

5.6 Class and Type Hierarchies

5.6.1 The Manifesto's demands

In object-oriented systems, the concept of inheritance permits objects to be organized in taxonomies in which specialized objects inherit the properties and operations of more general objects. Similar classes of objects which share properties and functions can be modelled by specifying a superclass, which defines the common part, and then deriving specialized classes (subclasses) from this superclass. This feature clearly provides powerful support for reusability and extensibility since the definition of new objects can be based on existing classes [Jeffery 92]. That is why the Manifesto demands inheritance as it is normal in object-oriented systems.

There are two possibilities of inheritance depending on the number of superclasses which may inherit their properties and functions to one subclass. Here the Manifesto demands only single inheritance, ie. every subclass has only one superclass.

5.6.2 Inheritance in ONTOS

As the DML of ONTOS is based on C++, inheritance is possible in ONTOS. As in C++ there is single inheritance and multiple inheritance (for details in multiple inheritance see Section 6.1).

The declaration of a class with name *classname* that inherits from class *superclassname* is:

class *classname* : public *superclassname*

The class *classname* gets all functions and data members from the superclass and then adds some data members and functions. It is possible to override functions of the superclass. When an object of the subclass calls this function, the function runs as it is declared in the class and substitutes the function of the superclass. (For more detail see Paragraph 6.6.2 .)

There is one special inheritance in ONTOS: every class that should be persistent must be derived from the class *Object*. When a class has a super-class that derives from the class *Object*, then all classes in this hierarchy are persistent.

In our video-shop example we used simple inheritance for the class "reservation" that inherits from class "arrangement". As the arrangement class is persistent the reservation class is as well. The declaration of both is as follows:

```
class arrangement : public Object
{
private:
        Date *from;
        Date *to;
        char *payment;
        char *transport;

public:

        arrangement();
        arrangement(APL* theAPL);

        Type *getDirectType();

        Date* get_from();
        void set_from(Date *the_Date);
```

```
        Date* get_to();
        void set_to(Date *the_Date);

        char *get_payment();
        void set_payment(char *the_payment);

        char *get_transport();
        void set_transport(char *the_transport);

        void dump();
};


class reservation : public arrangement
{
        char *the_video;
        Reference has_reservationer;

public:

        reservation(char *video_name);
        reservation(APL* theAPL);

        Type *getDirectType();

        char* get_video();

        member* get_member();
        void set_member(member *the_member);

        void dump_reservation();
};
```

Every object of Type reservation contains an object of Type arrangement (here called subobject). The subobject is created before the comprising object. The superclass constructor calls the subclass constructor. In the same order these objects are activated when they are retrieved from the database. So every function of the superclass can be called from an object of the subclass, as long as they are declared as public or protected. In our example all functions of class arrangement can be used within objects of class reservation or rental. Both subclasses have some additional methods and data members.

In contrast to C++ ONTOS does not allow virtual classes as superclasses [Ontos 92a]. But all the other features are the same.

5.7 Overriding, overloading and late binding

5.7.1 The Manifesto's demands

In some cases it is useful to have the same name for more than one operation.

Overriding is one form. The subclass contains a method with the same name as the method in its superclass. An instance of a subclass will execute the method defined in the subclass, when it is called with that function name.

Overloading is a more powerful form which contributes very much to get the writing of more legible program code. It permits the use of the same name (independent of any type- subtype relationship) to denote different functions. Ambiguities are resolved by the system examining

a syntactic facility which permits programmers to use the same name for different implementations of similar operations. An often used example is as follows: one may wish to use the same name for operations to add two integers, two real numbers or even two character strings. An add operation on variables v1 and v2 is invoked by the call:

add (v1,v2)

and the system decides which function to apply by examining the types of v1 and v2. Ambiguities may be resolved either at compile time or at run-time.

The Manifesto demands late binding which is the process, by which the version of the operation to be applied is determined at run-time or at least delayed.

5.7.2 Overriding, overloading and late binding in ONTOS

ONTOS has all the demanded features of Section 5.7.1, as they are already fulfilled by C++, which the DML is based on.

In our video-shop application we have used *overriding*. The class arrangement has a method called dump() which prints out the values of all data members. The reservation class also has a method called dump(), but we have redefined it. After that it did not only print out the values of the (inherited) data members, but also the name of the video and its copy number, which the reservation is made for.

(The method dump() is as well implemented in the class person, rental, member, video, video copy.)

We have also used *overloading* in our application program. Our database has three entries. From there you have to follow references to find objects of other classes. Every object of these three classes has a unique name. The "entry classes" are: member, video and video copy.

When a member gives his number, we just concatenate it with ^mem. So a member with a member number 123 is represented by an object with the name 123^mem.

A video object is identified by the video title and the medium (t for tape, l for LDLaserDisk): *title^medium*, so for example rainman^t.

A video copy object is identified by the video title, the medium and the copy number: for example rainman^t^2

To simplify the program code we have written 4 concatenate functions which create the object name from the given elements:

```
void concatenate(char s[], int i,char t[]);
void concatenate(int i,char s[],char t[]);
void concatenate(char s[], char c,char t[]);
void concatenate(char s[], char s2[],char t[]);
```

So within the program we just use concatenate, but do not care which of these functions actually runs.

Late binding is provided through the mechanism of virtual member functions. A virtual routine is provided with a definition in its original class, but may be redefined in descendant classes and it is the responsibility of the run-time system to find the appropriate version for each call of the virtual function.

In the predefined classes there are a lot of virtual functions, but the user can also write his own virtual functions. For example the predefined class Entity has the virtual function getDirectType:

```
virtual Type* getDirectType( ) = 0;
```

It returns the object's Type (Type is a class which represents C++ class definitions). It has to be redefined in every persistent class definition. This requirement ensures that an object's Type is always known at run-time. So, for example, in the member class:

```
Type *member::getDirectType()
{
        return memberType;
}
```

(memberType is set to *memberType = (Type *)OC_lookup("member")* in the main program.)

The use of the virtual function can be seen in the following part of our implementation:

```
member *my_member;
char mem_obj_name[40];
concatenate(member_nr,"mem",mem_obj_name);
Entity * the_entity = (Entity*) OC_lookup(mem_obj_name);
if (the_entity == NULL)
{
        cout << "member does not exist";
        return;
}
if ((the_entity->getDirectType()) == memberType)          // here is the call for the virtual function
{
        my_member = (member*) OC_lookup(mem_obj_name);
}
else
{
        cout <<"error : wrong object retrievement!";
        return;
}
```

When an object is retrieved from the database by name, we cannot be totally sure that it really has the Type we expect. Therefore we first retrieve it as an Entity and examine its Type by using the getDirectType function which then runs the implementation of the getDirectType function as defined in the member class. Then we can create an object of the class member with the retrieved values and so we do not risk to work with a wrong Type. ONTOS would not notice if the retrieved object is put in a wrong Type and would work then with the Bits and Bytes from the database as if they were of the right Type which is not very sensible.

5.8 Computational completeness

5.8.1 The Manifesto's demands

In sharp contrast to programming languages, traditional database query languages usually impose very severe restrictions on the kind of computation that can be performed. As a result, application programs must be implemented in a general-purpose language (host language) while access to data is realized via declarative data sublanguage, like SQL. As a consequence, data has to be passed between these two languages. Since both languages usually differ semantically, as well as structurally, such transformations may lead to a loss of information. This problem is known as impedance mismatch.

To avoid this, the Manifesto demands that one can express any computable function, using the DML of the database system. It does not need to be a new programming language for the database, but computational completeness can be introduced through a reasonable connection to existing programming languages.

5.8.2 Computational completeness in ONTOS

In ONTOS, the DML is the object programming language C++. As the object programming language is also the host programming language, programmers code only in one language. Retrieval of objects is done directly into and out of the host programming language, so no transformation of the object structure is needed.

C++ is computational complete. So ONTOS is capable of handling complex mathematical manipulations of data without a loss of information on the way between the database and the host language.

5.9 Extensibility

5.9.1 The Manifesto's demands

Within the OODBS there exist already some system-defined types. The user uses these types, but can also define new types. The user should not notice the difference between system defined and user-defined types, although there might be a difference in how the system supports them, but the user should not be aware of this.

The Manifesto does not demand that the user can extend the type constructors (e.g. sets, lists).

5.9.2 Extensibility in ONTOS

ONTOS does not provide a lot of system defined types. There are the primitive types like integer, character, float. Moreover, ONTOS has some predefined pointers and which are mainly pointers to the simple objects and there exists a date type. But all these cannot be made persistent by its own, only when using a constructor.

There is no difference between using user defined types or system defined types. The newly created types have the same status as the existing ones in our experience. So the extensibility demand is fulfilled.

5.10 Secondary storage management

5.10.1 The Manifesto's demands

The success of database systems, of course, largely relies on their ability to provide fast access to objects in the database. This can be supported through a set of mechanisms.

- index management: Indexing is well-known from conventional database systems where a database index consists of a set of index entries that are stored on disk, one index entry for each row existing in a table specified and responsive to future updates. Index entries look like rows of a table with two columns: the index key, consisting of the concatenation of values from certain column values in the row, and a row pointer to the disk posi-

tion of the row from which this specific entry was constructed. In object-oriented database systems the question is whether, with respect to encapsulation, one should index on the structure of the objects of a class (neglects encapsulation) [Jeffery 92].

- data clustering: The goal of clustering is to reduce the number of disk I/Os for object retrieval. The unit of data transferred from disk is a page instead of an individual object. If two objects are clustered on the same page, it will only take one disk I/O to access both objects successively. The second object is actually prefetched when the first one is accessed. If the page size is larger, more objects can be clustered on a page and one disk I/O can access them all.
- data buffering: Usually a buffer manager maintains a buffer of page frames and attempts in that buffer data that are likely to be accessed again soon. Transactions must issue a request to the buffer managing subsystem to load an item of data into the buffer before the transaction can access it. When the transaction is finished with the data it informs the buffer manager that the space occupied in the buffer by the data may be overwritten. A data item is thus guaranteed to remain in the buffer while it is in use.
- access path selection and query optimization: The goal is to get the result of a query very fast. The system should determine the best way to approach the database and execute the query over the database. It may make use of information in the database or knowledge of the whereabouts of particular data on the network to optimize the retrieval of a query.

These mechanisms are demanded by the Manifesto, but they should be invisible to the user. The application programmer should not have to write code to maintain indices, to allocate disk storage or to move data between disk and main memory. The programmer should work on a logical level of the system independent of the physical level below.

5.10.2 Secondary storage management in ONTOS

In ONTOS the server controls the physical storage manager that actually stores and retrieves objects to and from disk. The servers use segments of units of transfer from/to secondary storage, thus providing segment-based prefetching and buffering.

When the user wants to read or store an object, he must first activate it. The activation process involves allocating memory, reading the entire segment into the client cache and copying the object into the memory. In the process, references are translated from their database forms to either direct references or abstract References. But the user does not need to care about these representations.

After modifying the object, the user deactivates it, which means it is written to the server cache and when the transaction is committed, it is written to the database.

The scope of the activation refers to how many objects are activated in a single call. Objects may be activated singly or in groups. A group activation activates all objects contained in one of the aggregate classes. It may consist of any user-defined group of objects.

The storage management is handled through the user-accessible storage manager classes:

- Standard Storage Manager
- Group Storage Manager

- In-Memory Storage Manager

Each storage manager controls the storage behaviour of the object assigned to. This relation is defined at the creation of the object. The three kinds of storage managers can coexist in the same application, managing different objects.

All three allocate and deallocate memory, activate and deactivate objects, set locks and release locks, cluster objects in the database, maintain and resolve references from objects under its control to other objects and delete objects from the database. These services are performed transparently.

The objects are clustered via either system default or application control. The storage managers perform application specific clustering by specifying the location of an object relative to another; then they store the new object in the same segment as the target segment. (With the same function the user can define in which area he wants to store his object.)

Each of the three storage managers has a special feature:

- The Group Storage Manager is optimized for handling a group of small objects (smaller than 500 Bytes each) when they are all needed in one transaction.
- The In-Memory Manager is optimized for non-persistent objects.
- The Standard Storage Manager is the default storage manager.

Moreover, ONTOS offers the possibility to access data members via indexing. The default behaviour of classify is not to create an index. But if the user wants to have one, he can get it. The user has to specify to create an unordered or an ordered index. The manual advises only to use indices on data members that have a simple type like integer or char*. Creation of an index allows fast access to the set of instances having a given value (a range of values) for the data member via an InstanceIterator.

As pointed out in the manual, ONTOS provides query optimization, but we could not test it.

So all demands of the Manifesto in this respect are fulfilled.

5.11 Concurrency

5.11.1 The Manifesto's demands

It is generally expected that user programs will attempt to read and write the same pieces of information at the same time. Doing so creates an access conflict for the data. That is why the Manifesto demands a concurrency control mechanism that is established to mediate between these conflicts and that does so by instituting policies that specify how read and write conflicts will be handled.

Usually a "sequence of operations" must be executed as a unit which is called transaction. Intermediate states, which exist after individual statements of an updating transaction have been performed, may be inconsistent. Therefore, to guarantee the consistency of the database, the Manifesto demands that transactions must be processed entirely or not at all (ie. transactions are atomic). All transactions must preserve the consistency and correctness of the data stored in the database. That is, the operations performed by a transaction should transform the database from one consistent state to another.

To give the user an understandable view of the database the effect of transactions must be that which he would be obtained if no other transactions were executed concurrently. The effect of executing several transactions concurrently, therefore, must be the same as if they had been executed serially in some order. Concurrently executing transactions whose effect is equivalent to that of some serial execution are said to be serializable. This, at least, is demanded by the Manifesto.

5.11.2 Concurrency in ONTOS

In ONTOS each access to the database has to be done within a transaction, which has to start with the command OC_transactionStart() and end either with OC_transactionCommit() or OC_transactionAbort(). With the start of a transaction the OODBS user defines the concurrency control. Whenever he wants to read or write from/to the database, he must acquire a lock for this object. ONTOS provides several locks, but in the sense of concurrency only ReadLock and WriteIntentLock are important. Whenever an object is activated, the user can determine which lock he wants for this object or just rely on the default lock, which depends on the concurrency protocol (see below). The locks are released at commit time at the earliest, which is usually the case, but the user cannot determine the point of time.

ONTOS supports the conservative, the time-based and the optimistic concurrency protocol. In the *conservative concurrency protocol* the object is checked for access conflicts when a lock for this object is requested. It enforces serialization. A process attempting to get a ReadLock or WriteIntentLock on an object WriteIntentLocked by someone else or attempting to get a WriteIntentLock on an object that already has a ReadLock, cannot access the object.

The *time-based concurrency protocol* is a middle alternative to conservative and optimistic control. It assumes (with some confidence) that all conflicts can be serialized, but checks periodically to be sure that the transaction does not continue uselessly with an undetected and irresolvable lock.

Under the *optimistic policy*, each transaction performs its updates on a private copy of data and the transaction is validated at commit time by ensuring that the original data is not also been accessed by a concurrently executing transaction. That is also the point of time when conflicts are detected. The optimistic policy provides the widest overall access to data and accepts a relatively higher risk of abort due to irresolvable conflict than either of the other policies. The optimistic concurrency protocol allows a ReadLock to be set on an object that has a WriteIntentLock if conflicting transactions can be serialized. The difference between the time-based and the optimistic policy is the point of time when possible conflicts are detected.

ONTOS has no variable which the OODBS user just sets to one of the 3 policies. The user has to set 3 object-handling protocols. Depending on the combination the user has chosen the concurrency protocol is one of the concurrency protocols described above or somewhere between them. Section 5.11.3. describes the procedure in more detail.

The ONTOS manual says that the transactions are atomic. We have made no other experiences in our tests with the example.

As explained above, ONTOS fulfils the concurrency demands of the Manifesto.

5.11.3 Defining concurrency protocols in ONTOS

With the start of the transaction the OODBS user defines 3 object-handling protocols which are global to the transaction.These protocols are

- A conflict detection protocol which is for identifying the conflicts arising from the current attempts to access an object.
- A conflict response protocol which is for responding to conflicts arising from attempts to lock an object for reading or writing
- A buffering protocol which defines how many objects are buffered on the client side before they are output to the server cache.

ONTOS offers two conflict detection protocols. (They help to find conflicts before the object copy comes into the client cache. This protocol does not prevent the user from changing a lock when an object copy is already in the client cache.):

1. RWConflict
   Under this protocol the only concurrent access allowed is to read on an object that is already ReadLocked. By default objects are activated with WriteIntentLock.
2. NoRWConflict
   This protocol maximizes overall concurrency across all applications accessing the database. It allows processes to obtain ReadLocks on an object that has already been WriteIntent-Locked. The readers of the object see an earlier version if they are serialized earlier than the writers to the object. A preemptive abort occurs if a reader and a writer of "object1" exchange roles for "object2" (deadlock!). Here objects are activated with Readlock by default.

Table 1 shows what happens when one transaction tries to lock an object that has already been locked by another transaction depending on the kind of lock and on the kind of conflict detection protocol.

Table 1: Conflict resolution of concurrent processes depending on conflict detection protocol and kind of lock

			new lock requester			
			NoRWConflict		RWConflict	
			ReadLock	WriteIntentLock	ReadLock	WriteIntentLock
lock holder	NoRW-Conflict	ReadLock	success	success / abort	success	conflict / abort
		WriteIntentLock	success / abort	conflict / abort	conflict / abort	conflict / abort
	RWCon flict	ReadLock	success	conflict / abort	success	conflict / abort
		WriteIntentLock	conflicts / abort	conflict / abort	conflict / abort	conflict / abort

The conflict response protocol defines what should happen when there is a lock conflict detected with the help of the conflict detection protocol. ONTOS provides a choice of two conflict response functions:

1. OC_waitOnConflict
   Waits until the lock is relinquished by the locking process to complete the database operation
2. OC_notifyConflict
   Raises the "WaitException" if there is a conflict. So the application can regain control after a lock conflict and can retry the database access or do other work.

The second conflict response function is used as below. If a conflict occurs the transaction raises the ExceptionHandler and then the transaction is aborted and the "else" part runs.

```
.....
ExceptionHandler lockError("WaitException");
if (lockError.doesNotOccur())
{
        .....
        OC_transactionStart(RWConflict,OC_notifyConflict);
        ..... here are the commands within the transaction
        OC_transactionCommit;
}
else OC_transactionAbort();
..... //this part runs when a confict occurs
```

When objects are put to the database, they must be transferred from the client to the server, usually over the network. This transfer can be made more efficient if the objects are buffered in client memory and transferred in groups. However, until a put is actually made to the server, the application cannot get any information on lock conflicts that could not be caught with the conflict detection protocol when the locks were requested for the objects in the database. Therefore, it is the best to group only a moderate number of objects into a single transmission. ONTOS leaves this decision to the user. It provides 3 buffering protocols:

1. OC_noBuffering
   Each put call results in an intermediate transmission to the server.
2. OC_defaultBuffering
   Objects are buffered and sent in small groups to the server (usually after 10 put operations, but the user can define this number).
3. OC_bufferUntilCommit
   Objects are buffered during the transaction and transmitted all at once when the transaction is committed. But if the buffer is exhausted, the system will make interim transmission.

Some combinations of the different protocols are known as concurrency control policies:

- Conservative concurrency control is achieved when the RWConflict detection protocol and OC_noBuffering is used.
- Time-based concurrency control is achieved when the NoRWConflict protocol and either the OC_noBuffering or the OC_defaultBuffering is used.
- Optimistic concurrency control is achieved when the NoRWConflict protocol is combined with the OC_bufferUntilCommit buffering protocol. Conflicts are realized only at commit or checkpoint time.

5.12 Recovery

5.12.1 The Manifesto's demands

The Manifesto demands software tools to implement recovery in the event of system failures. These tools have to ensure atomicy and to avoid inconsistent data states. So in case of hardware or software failures, the system should recover, ie. bring itself back to some coherent state of data.

5.12.2 Recovery in ONTOS

ONTOS maintains special repositories called journals. Every area has its journals. The journals record the history of each transaction which has updated the area since the last back-up copy of the area was made. The journal saves which transaction has updated or created which object, its old value and the new value. Moreover key points in the progress of transactions, such as their start and end times, are stored and the point at which a transaction commits is recorded. When all changes of a transaction have been recorded in the journal, the transaction issues the commit message which makes the server of the area to transmit the changes from the journal to the area. The server reads from the journal at two specified times only:

- when it updates the area
- when it is first activated (if recovery is required)

In an event of a hard crash the DBATool can be used to recover a single area or all the areas in the logical database and roll a back-up copy of the area / areas forward to its state prior to the crash by replaying all the journals from the area / areas. That is why back-up copies should be made occasionally. It is essential to backup all databases that use the same kernel area as one unit. With this back-up one must also make a copy of the registry. So if the it has been corrupted one just restores the old registry file and replace the current registry with the old one.(For more details about registry, kernel area, etc, see Section 3.)

The journals are implemented as files and are written to the same directory that contains their area file. The name of a journal file is composed of three parts <areaname>.JRN.<number>. The "JRN" suffix identifies the file as a journal file. The <number> suffix is used by the server and the DBATool to determine which journals to replay and in which order.

Due to our experiences on this field ONTOS fulfils the Manifesto's demands for recovery.

5.13 Ad hoc query facility

5.13.1 The Manifesto's demands

The Manifesto demands a service which allows the user to ask simple queries to the database. This has not to be in form of a query language, but it should have the functionality of a query language. One way is to support it by the data manipulation language.

The Manifesto has some criteria which have to be fulfilled by the query facility:

1. It should be high level, ie. one should be able to express non-trivial queries concisely;
2. It should be efficient and optimize the query itself;

3. It should be application independent.

5.13.2 Query facility in ONTOS

ONTOS provides Object SQL which allows some of the basic SQL commands such as *select, from, where, and, or* and standard Boolean and relational operators such as *is in, is not in*, etc. ONTOS allows queries to be made over the properties (data members) and procedures of persistent objects. Object SQL is implemented with a single class called QueryIterator which gets the SQL expression as a string as its argument and which allows queries to be stored as objects in the database. Each instance of the class represents a particular query. The results of the query are obtained by calling the yieldRowString() member function. Each call returns the next row of results. The following is a part of the query application in which we ask for the first name and the town of the person's address whose second name is 'Smith'.

```
char the_output[200];
QueryIterator *my_iterator = new QueryIterator("select p.first_Name,p.town  from person p where
p.second_Name=\"Smith\";");
while(my_iterator->moreData())
{
        my_iterator->yieldRowString(the_output, 200);
        cout   << " "
               << the_output
               << "\n";
}
```

The query above accesses the data members of the class person which are declared as private and usually only accessible via a function of the class. But Object SQL ignores these access permissions and prohibitions. Actually the query would have been:

```
QueryIterator *my_iterator = new QueryIterator ("select p.get_first_name(), p.get_town() from per-
son p where p.get_second_name()=\"Smith\";");
```

This query has the same results as the one above.

We have shown how to access rows of the query's result, but it is also possible to break the row into columns (with the help of the yieldRowIterator() function) and so the user is able to programmatically access a property returned as a result of an SQL query. This is especially usefully when the result contains objects. But in our example

```
localNameGenerate((Entity*)my_argument, the_output, 199,,)
```

returns the value of the string, but if it would be an object it would return the local name of it. If it does not exist, ONTOS creates it.

```
char the_output[200];
QueryIterator *my_iterator = new QueryIterator("select p.first_Name from person p where
p.second_Name=\"Smith\";");
cout << "Programmatic version of columns:\n";
while (my_iterator->moreData())
{
        Iterator *my_row_iterator = my_iterator->yieldRowIterator();
        while (my_row_iterator->moreData())
        {
                Argument my_argument = (*my_row_iterator)();
                cout << OC_localNameGenerate((Entity*)my_argument, the_output, 199,,) << " ";
        }
        cout << "\n";
}
```

Query iterators support recursive and hierarchical queries. The FROM clause in Object SQL accepts any argument that evaluates to a collection of objects in addition to class names. The SELECT clause accepts property names as well as member function invocations and navigational style property chain.

Scrutinizing ONTOS for the demanded criteria of the Manifesto, we have discovered the following.

ONTOS provides a high level query facility with OSQL as the query itself is formulated as a string. But to run the query a whole application has to be written and compiled. We could not check if the formulation of the queries lends itself to some form of query optimization, but the manual says ONTOS does so. But as the queries are conducted against extensions of classes and aggregates the last criteria of the Manifesto's demand is not fulfilled. It is not self-evident that all classes have extensions. The user can determine which classes have extensions. Moreover, any functions which are to be used as part of a SQL query must not be declared inline.

So the ONTOS query facility does not fulfil all demands of the Manifesto for query facility.

In order to look at the data saved in the database ONTOS provides a browser. The browser cannot look at all instances, ie. at instances of multi- inherited classes. It is very easy to use, but cannot answer queries.

6 Comparison of Optional Features

6.1 Multiple inheritance

Multiple inheritance permits a new type to be derived from a number of other classes. This is used in the video-shop application for the Type video which has two parent classes namely videomovie and videomedium. The Type videomovie has information about the title of a movie, main star, the duration of the movie in minutes and the kind of movie (ie. humorous film, thriller, etc.). Whereas the Type videomedium represents the material which the film is recorded on and is described by the kind of medium (tape or LDLaserDisk) and the length of maximal time that medium can record (in minutes). A video consists of the vidoemedium and the movie, as a computer consists of hardware and software.

Here is the class description containing these three classes:

```
class Videomovie : public Object
{
protected:
        char* titel;
        int duration;
        char* main_star;
        char* kind_of_movie;

public:
        Videomovie(char* the_object_name,
                        char* the_titel,
                        int the_duration,
                        char* the_main_star,
                        char* kind_of_movie);
        Videomovie(APL* theAPL);

        Type* getDirectType();

        char* get_titel();
        int get_duratin();
        char* get_main_star();
        char *get_kind()

        void dump();;
};

class Videomedium : public Object
{
protected:
        char media;
        int length;             // t = tape, l =LaserDisc

public:
        Videomedium( char the_media,int the_length);
        Videomedium(APL* theAPL);

        Type* getDirectType();

        char getmedia();
        int getlength();
};

class Video : public  Videomovie, public Videomedium
{
private:
        int number_of_copies;
```

```
        Reference is_representative_of;

public:
        Video(char* the_object_name,char* the_titel, int the_duration, char* the_main_star,char*
kind_of_movie,char the_media='t',int the_length=120,);
        Video(APL* theAPL);

        void* operator new(OC_size_t sz);
        void* operator new(OC_size_t sz,APL* the_APL);
        void* operator new(OC_size_t sz,StorageManager* sm,Type* t);
        void operator delete(void* v);

        virtual void* startAddress() {return this;}
        Type *getDirectType();

        void putObject(Boolean deallocate = FALSE);
        void deleteObject(Boolean deallocate = TRUE);

        int copies();
        void inkrement_copies();
        void dekrement_copies();

        void set_represents(Dictionary *dic);
        Dictionary* get_represents();

        void dump();
};
```

The definition of persistent classes with multiple base classes is a little bit more complicated than the definition of a persistent class with one base class. There are some additional requirements and some functions must be redefined.

As for every persistent class the user must define the special constructor that is used when instances of the class are retrieved from the database. This constructor takes the argument typed as "APL*". It must call the activation constructors of all base classes.

```
Video::Video(APL* theAPL) : Videomovie(theAPL), Videomedium(theAPL)
{
        cout <<"\n Activating Video\n";
}
```

A problem is that all persistent classes inherit directly or indirectly from the class "Object". Hence, the user must define one of the base classes as the primary (first) persistent base class. All instances of the multi-inherited class use the instance of the class "Object" of the primary persistent base class to manage storage and persistence. In our example we define Videomovie as the primary base class. Through this base class we pass the object name and call the directType() function, which sets a pointer to the persistent representation of a class (a pointer to the Type object of this class):

```
Video::Video(char* the_name, char* the_titel, char* the_main_star, char* the_director,
                int the_length, char the_media)
: Videomovie(the_name, the_titel, the_main_star, the_director), Videotape(the_length, the_media)
{
        Videomovie::directType(VideoType);
        number_of_copies = 1;
        is_representative_of.initToNull();
        cout << "Creating Video\n";
}
```

The new operators and the delete operator must also be redefined and simply call the corresponding operator on the primary persistent base class. The putObject() and deleteObject() functions call as well the corresponding functions each of class Video's persistent base classes.

```
void Video::putObject(Boolean deallocate)
{
        Videomedium:: putObject(FALSE);
        Videomovie:: putObject(FALSE);
        if (deallocate) delete this;
}
```

When we make a set of all Videos that are on tape, for every member of the set a reference is stored to find the members. The reference is stored within the set to the start of the first persistent base class of the instance. Even if we have a set of Videomedium, we can insert a Video instance into our set, because after all, a Video is a Videomedium. When we retrieve our element from the set, we must do some casting to get the set.

ONTOS does not allow virtual persistent base classes for multiple inheritance. Private persistent base classes are not allowed either. The persistent base class must be public. But the base classes do not need to be persistent. However, in order for the new class to be persistent, one of its base classes must be persistent.

When the persistent base classes have some members that have the same name, ONTOS does not know, which one to chose and sends an error message during the compilation. But it is possible to define the base class which the member should be taken from.

6.2 Type checking and type inferencing

In conventional typed languages, the compiler assigns a type to every expression and subexpression. However, the programmer does not have to specify the type of each subexpression. Type information need only be placed at critical points in a program, and the rest is deduced from the context. This deduction process is called type inference. Typically, type information is given for local variables and for function arguments and results. The type of expressions and statements can then be inferred, given that types of variables and basic constants are known. Type inferencing reduces to type checking when there is so much type information that the type inference task becomes trivial [CarWe 85].

The Manifesto does not prescribe the degree of type checking the system will perform at compile time. But the best situation would be if all type errors were detected at run-time and a compiled program does not produce any run-time type errors.

The Manifesto also leaves it to the system designer if the system offers type inferencing. Again the same principle: the more the better. It would be desirable if only the base types had to be declared and the system inferred the temporary type.

The DML in ONTOS is based on C++ which is a strongly typed language. So it offers the ability to determine the type compatibility of all expressions representing values from the static program representation at compile time as long as there is no interaction with the database. Both the argument list and the return type of every function call are type checked during compilation. If there is a type mismatch between an actual type and a type declared in a function prototype, an implicit conversion will be applied if possible. If an implicit conversion is not

possible or if the number of arguments is incorrect, a compile-time error is issued. The function prototype provides the compiler with the type information necessary for it to perform type checking at compile-time.

These conversions are only for simple types: from int to double, from C++ int to Integer which represents values of C++ primitive data type int in ONTOS, etc.

ONTOS makes type checking for data going into/out of the database against "classified" types. However, on retrieval via "OC_lookup()" it relies on the programmer casting the returned data to the correct type, so in this case there is a potential weakness. ONTOS prints out an error message at compile-time for the following command:

```
account * a = (member *)OC_lookup("123");
```

But is does not notice when a retrieved object of the type video (ie. video with title "rainman" which is a tape) is returned into an object of the type member and does not raise any error messages (neither at compile-time nor at run-time):

```
member * m = (member*) OC_lookup("rainman^t");
```

The user has to check the type of the retrieved object and then cast it to the correct type (for further detail see Section 5.7.2).

6.3 Distribution

In a distributed database the data are held on a network of computers across different sites (or nodes) which are geographically remote from each other. Each site holds a partition of the database.

The Manifesto does not prescribe if the database must be distributed or not.

ONTOS manages a database that may be physically distributed over a local area network. But the nodes must be of the same hardware family and must be running the same operating system [Ontos 92a].

The database is subdivided in so called areas which are physical files where the objects are stored. They can exist anywhere on the network. These files are configured by the user before any application runs.

ONTOS distinguishes between the logical database and the physical database. The logical database consists of areas and must include the kernel area which consists of the metaschema classes. The physical database consists of a kernel area and all the files containing areas that are rooted in the kernel area. All logical databases that share the same kernel area are part of the same physical database. An area can belong to more than one logical database within the physical database.

The locations of the area files are transparent to the user and so the user can get objects from the database or put them into the database without having to specify the identity and/or location of the physical file where the objects are stored. But he has the possibility to do so. Moreover, objects may reference other objects within the network without knowledge of the network's actual configuration or the physical location of the objects at any given time.

6.4 Design transactions

The notion of a transaction in a design application can be very different to a business data processing application. Design applications (ie. in CAD) are typically large and complex and the transactions are usually of long duration. It must be expected, therefore, that for cooperative design activities, information will be shared before the completion of a design transaction. Hence, in some instances the traditional notion of serializability as a correctness criterion for concurrently executing transactions may be too restrictive, as it insists that concurrent execution of transactions must produce results that are equivalent to some serial execution of those transactions. This precludes data sharing during transaction execution.

The most optimistic approach to concurrency control which ONTOS offers is that several transactions can read one object while another transaction writes on this object. Each transaction performs its updates on a private copy. When the transaction commits, the updates are made to the database if the transaction is serializable with other transactions. (For more detail see Section 5.12.)

Much greater concurrency is possible if multiple versions of an object can exist. Then many users can simultaneously access multiple versions of an object without conflict. Thus, a number of OODBS handle concurrency using a check-in / check-out approach. When a user wishes to change an object, a version of that object is checked out into the private workspace. This effectively sets a write-lock on that version of the object, disabling other users from attempting to change that version. Once the change has been made, the user checks-in the object, which creates a new object version. Unfortunately, ONTOS does not have the check-in / check-out approach and does not support versions of an object.

The long lifetime of design transactions means that traditional approaches to database recovery based on transaction boundaries could result in a great deal of work being lost when a transaction failure occurs. ONTOS offers checkpointing to help.

In order to commit all changes made to the database during a transaction, ONTOS has a function called OC_transactionCheckpoint(). It does not terminate the transaction or release the transaction locks, but it allows long transactions to be committed in stages and is a way of reducing the amount of work exposed to a possible transaction abort. Once a checkpoint completes successfully, all objects that have been committed are safe even if the transaction is subsequently aborted. In the event of system failure a transaction can be restored to the last checkpoint. However, this technique may lead to significant loss of data unless checkpoints are frequent.

Moreover, ONTOS offers nested transactions. A nested transaction is a transaction that begins and ends between the beginning and ending of another transaction. Changes to the database made in a nested transaction are contingent on the successful commitment of all its ancestral transactions. Aborting any of its ancestor invalidates all its changes. If a nested transaction aborts, the database state seen by its parent is the same as it was immediately prior to starting the nested transaction. In ONTOS nesting may occur to a depth of 31 levels.

So with the help of nested transactions it is possible to group a subset of changes made by a top-level transaction in contrast to a usual transaction. Between the beginning and the ending of the transaction all incremental changes are made or none of them. So it is possible to abort changes selectively. This is very useful for complex applications.

So all in all, ONTOS offers quite a lot for design transactions, but there are also some important features like a check-in/check-out approach missing.

6.5 Versions

Many database applications require the capability to create and access multiple versions of an object. Important uses of versioning are to be found in databases underlying tools for Computer Aided Software Engineering (CASE), and Computer Aided Design (CAD). It is widely recognized that version control is one of the most important functions in environments in which users need to generate and experiment with multiple versions of an object before selecting one that satisfies their needs. It also helps to keep track of the evolution of design since objects may store their version history. In the event that a design appears to be faulty at any stage, it is then possible to rollback the design to some valid data state.

ONTOS does not support versions in our release.

7 Conclusion

We have presented a detailed evaluation of the object-oriented database system ONTOS Version 2.2 with regard to the features the Manifesto demands of an OODBS. As these were not factual enough, we had to put them in concrete terms. Then we scrutinized ONTOS to the expanded demands by writing an application. We implemented a video-shop software which is a well known test example in the United Kingdom. We could test most of the demanded features with an application, but some were not checkable and so we had to rely on what the ONTOS manual says.

An objective rating of our results on a scale of 0 (poor) to 5 (totally fulfilled) is shown in Table 2. The scale should only help to roughly and clearly see the outcome of our studies. As in the Manifesto the features are divided into mandatory and optional features. We did not evaluate the open features because they were left as open choices by the Manifesto's authors who did not agree on the kind of their realization. The open features do not contain any concrete expectations to a system.

Table 2: Evaluation of ONTOS with regard to the Manifesto's demands

Manifesto features		ONTOS
mandatory	complex objects	5
	object identity	4
	encapsulation	5
	types and classes	5
	class or type hierarchies	5
	overriding, overloading and late binding	5
	computational completeness	5
	extensibility	5
	persistence	5
	secondary storage management	5
	concurrency	5
	recovery	5
	ad hoc query facility	3
optional	multiple inheritance	5
	type checking and type inferencing	3
	distribution	3
	design transactions	4
	versions	0

Table 2 indicates that ONTOS satisfies the demands of the Manifesto to a very high degree. In particular, eleven of the mandatory features are totally fulfilled and two of the mandatory features (object identity, ad hoc query facility) are substantially realized. Furthermore, most of the optional features, the so-called "goodies", are also satisfied.

The mandatory feature "object identity" is fulfilled in so far as ONTOS gives each object a unique identity and manages it. But the Manifesto also demands operations like object assignment and object copy which are not directly offered by the system. The user has to write the code within predefined templates. The mandatory feature "ad hoc query facility" is partially fulfilled since it exists, but it is not database independent and there is no high level query language.

As far as the optional features are concerned, the requirements for versions do not exist. Distribution is provided, but only over a network of homogeneous workstations. Multiple inheritance is totally fulfilled. Design transactions, type checking and type inference are present, but could be improved.

As can be seen, ONTOS does well with regard to the mandatory features. However, this result cannot be qualified easily, as similar evaluations for other products are not available. ONTOS may just be a very good system. On the other hand, there has been much time between the Manifesto's publication and the delivery of ONTOS Version 2.2. As the Manifesto has become a strong reference paper for the industry, the good results of ONTOS may be nothing unusual for currently available OODBSs. An equivalent statement applies to the mentioned deficiencies in the optional features. It could not be clarified in this work whether some of them are missing because of lower importance, or whether their realization turned out to be difficult for this system.

Strictly speaking, ONTOS is not an OODBS in the Manifesto's sense. However, when choosing a product the user must decide on the importance of each criterion for a particular application.

8 References

[AASW 94] K. Abramowicz, J. Alt, H. Schreiber, M. Wallrath : *"Evaluierung objektorientierter Datenbanksysteme"*, FZI- Publikation 3/94, Forschungszentrum Informatik an der Universität Karlsruhe, Germany

[AASW 95] K. Abramowicz, J. Alt, H. Schreiber, M. Wallrath : *"Object-Oriented Databases: The Struggle for a Dominant Market Share — A Critical Evaluation of the Leading Products "*, FZI- Publikation 1/95, Forschungszentrum Informatik an der Universität Karlsruhe, Germany

[ABDB 89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik : *"The object-oriented database system manifesto"* in Deductive and Object-Oriented Databases. Proceedings of the First International Conference (DOOD89), p. 223-240

[AtBuM 88] M. Atkinson, P. Buneman, R. Morrison : *"Binding and Type Checking in Database Programming Languages"* in The Computer Journal, Volume 31, No 2, p. 99-109

[Brown 91] A. Brown : *"Object-Oriented Databases — Applications in Software Engineering"*, McGRAW-HILL Book Company, 1991

[CarWe 85] L. Cardelli, P. Wegener : *"On Understanding Types, Data Abstraction, and Polymorphism"* in ACM Computing Surveys, Volume 17, No 4, p. 471-519

[Halpin 94] Halpin, T. A. :"Object Role Modelling — NIAM and Beyond" Tutorial, ER94 Conference, The University of Manchester Institute of Science and Technologie, Manchester, UK

[HuPa 93] A. Hurson, S. Pakzad : *"Evolution and Performance Issues"*, in Computer, February 1993, Volume 26, No 2, p. 48- 60

[Jeffery 92] K. Jeffery : *"Expert Database Systems"*, Academic Press, 1992

[Kfoury 82] A.J. Kfoury, Robert N. Moll, Michael A. Arbib : *"A programming approach to computability"*, Springer-Verlag New York Heidelberg Berlin, 1982

[Lippm 92] St. Lippman : *"C++ Primer"*, 2nd Edition, Addison-Wesley Publishing Company, 1992

[Ontos 92a] Ontos, Inc. ONTOS DB 2.2, Developer's Guide, 1992

[Ontos 92b] Ontos, Inc. ONTOS DB 2.2, First Time User's Guide, 1992

[Ontos 92c] Ontos, Inc. ONTOS DB 2.2, Reference Manual, vol.1. , 1992

[Ontos 92d] Ontos, Inc. ONTOS DB 2.2, Tools and Utilities Guide, 1992

[Solov 92] Valery Soloviev : *"An Overview of Three Commercial Object-Oriented Database Management Systems: ONTOS, ObjectStore, and O_2"* in Sigmod Record Volume 21, No 1, March 1992, p. 93-104