Ode 2.0 User's Manual

A. Biliris
N. Gehani
D. Lieuwen
E. Panagos
T. Roycraft

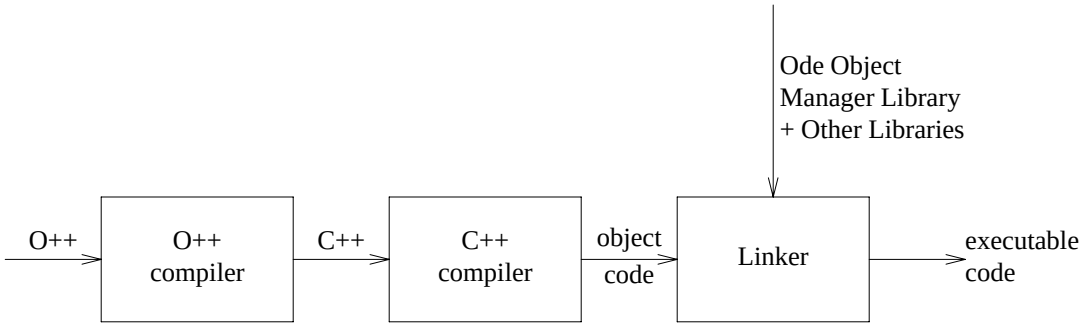
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

Ode is a database system and environment based on the object paradigm. The primary interface for the Ode database is the database programming language O++, which is based on C++. A few facilities have been added to C++ to make it suitable for database applications. O++ provides facilities for creating persistent objects which are stored in the database and for querying the database. Future releases of Ode will support sets, associating constraints and triggers with classes, large objects, versioned objects, and concurrency control. The Ode database is based on a client-server architecture. Each application runs as a client of the Ode database. The storage manager used by Ode is EOS.

2. Ode Release 2.0

Ode 2.0 is available to ''friendly'' users. Ode 2.0 consists of the O++ compiler and the object manager manager library. Database applications are written in O++. The O++ compiler produces C++ code, which is then compiled with the C++ compiler:

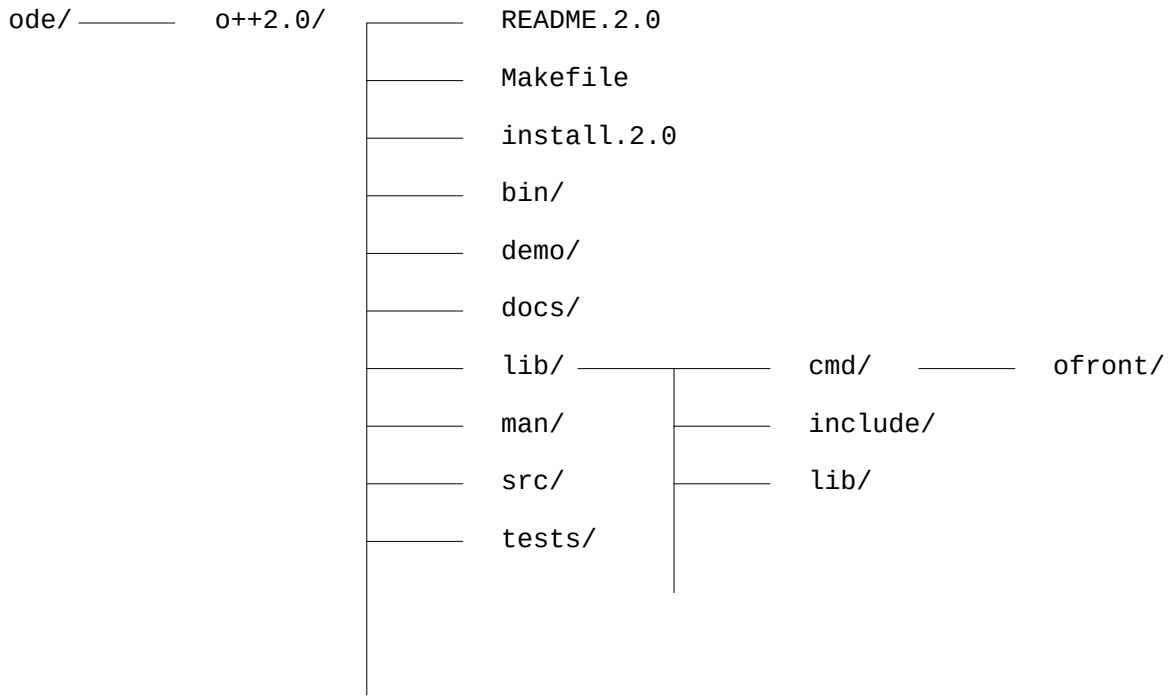


3. Requirements for using Ode

1. A C++ compiler that conforms to C++ 2.1 or higher.
2. Sun Sparcstation (or a Sun with a Sparc processor) running SunOS 4.1.x.
3. The SunOS must be configured to support shared memory, semaphores, and message queues.

4. Installing Ode

The Ode system is organized as follows:



4.1 Installing Binaries

Create a directory and unbundle the tar file (tape) in that directory.

`tar xvf` *device-or-file*

If the file ends in `.Z`, then before unbundling with `tar`, uncompress it as

`uncompress` *compressed-file*

After unbundling the tar file, read the file `README.2.0`, make the changes specified in the file `install.2.0` and type

`./install.2.0`

4.2 Installing Source and Compiling Ode

Create a directory and unbundle the tar file (tape) in that directory.

`tar xvf` *device-or-file*

If the file ends in `.Z`, then before unbundling with `tar`, uncompress it as

`uncompress` *compressed-file*

After unbundling the tar file, read the file `README.2.0`, make the changes specified and recompile as specified.

5. Environment Variables

So as to invoke the client and server initialization commands, server startup commands, and the O++ compiler without explicitly specifying the full path, add the following to your `PATH` variable:

ode-directory-path`/o++2.0/bin`

Also, add

ode-directory-path/o++2.0/man

to your MANPATH variable.

Ode databases reside in database areas which may be UNIX® files or raw disk partitions. A database area may contain one or more databases. Within programs (applications), databases may be specified by providing a full path name to the database area followed by the database name; alternatively, a database may be created or assumed to exist in an optional default area. To use a default area, the environment variable ODE_DEFAULT_AREA must be set to the path to the default area.

Suppose, for example, that databases are to reside in the database area DATABASES under your home directory. To do this, perform the following steps:

1. Create area DATABASES under your home directory using the command odeareaformat.
2. Next, if you use csh or tcsh shells, place the following instruction in your .cshrc file:

   ```
   setenv ODE_DEFAULT_AREA ~/DATABASES
   ```

   If you use sh or ksh shells, place the following instructions in your .profile file:

   ```
   ODE_DEFAULT_AREA = ~/DATABASES; export ODE_DEFAULT_AREA
   ```

6. Database Area, Server, and Client Initialization

6.1 Initialization (Configuration) Files

Ode uses three initialization files for storing parameters used for the server, clients, and for formatting areas. Initialization files are located in the directory .eos (created automatically by the initialization commands) located in the user's home directory.

Databases are located in database areas which are created (formatted) using the information stored in the file ~/.eos/formatrc file. The server reads the file ~/.eos/serverrc when starting up. Clients read the initialization file ~/.eos/clientrc before starting.

The following commands are provided for creating the initialization files:

- make_server_init
- make_client_init
- make_format_init

These commands do not take any arguments.

The server and format initialization commands must be run on the host machine that will run the server. The client initialization command should be run by each user who intends to run a client. After these initialization files have been created, they can be edited to change the information stored in them.

If the server is running on a machine different from the one specified as the value of the variable EOS_SERVER_HOST_NAME in file ~/.eos/clientrc (say as specified during initialization) then the value of EOS_SERVER_HOST_NAME must be changed appropriately.

The log records generated by a transaction (files with the prefix priv_) are stored in the directory specified by the variable EOS_PRIVATE_LOG_DIR in the file ~/.eos/clientrc. By default, this directory is assumed to be \tmp. An alternative directory can be specifed changing this variable appropriately.

6.2 Creating Database Areas

Database areas are created (formatted) with the command odeareaformat:

® UNIX is a registered trademark of USL.

`odeareaformat -l` *database_area* `-o`

The area must be explicitly specified (even though the environment variable `ODE_DEFAULT_AREA` may have been set). The server should not be not running when `odeareaformat` is executed. This command will overwrite an existing area with the same name, if one exists. See the man page for details about `odeareaformat`.

For example, a database area `test1` may be created in the directory `/usr/nhg/odedb` as follows:

```
odeareaformat -l /usr/nhg/odedb/test1 -o
```

6.3 Starting the Server

Before client programs can be run, the Ode server must be started by executing the command

```
odeserver
```

Only one server can be running on a machine. The server is typically run in a separate window (in the foreground) because the server accepts commands from `stdin` and because it prints messages.

The two server commands that are used most often are `checkpoint` and `shutdown`. The `checkpoint` command causes the server to flush its buffers and write out all committed transactions to disk. The `shutdown` command causes the server to terminate. See the man page for details about `odeserver` and its commands.

6.4 Client Programs

Client programs (applications) are executables produced by the O++ compiler.

7. Changes in Ode 2.0 with respect to Ode 1.1

1. Only objects of classes for which a persistent forward declaration has been given before the class definition is encountered can be made persistent. For example.

   ```
   persistent class employee;
   class employee {
       ...
   };
   ```

   Previously, only objects of classes that followed the `#include` directive

   ```
   #include <ode.h>
   ```

   could be made persistent. The above use of the `#include` directive to specify classes whose objects could be made persistent was abandoned because this approach was error prone and inappropriate from software engineering considerations. Note that the header file must still be included prior to using any of the O++ facilities.

2. The database must be created, opened, closed, or removed from outside the transaction body.
3. The large object class interface header file, `lo.h`, has been renamed to `large.h`.
4. Object versioning is now supported.
5. Ode now supports full-fledged transactions. Three flavors of transactions are supported: update, read-only, and hypothetical transactions.
6. Client-server architecture is supported.
7. 2-version 2-phase locking is used.

8. O++ Database Facilities

We now give a brief description of O++ and how O++ programs are written. We will only describe the facilities that O++ adds to C++. The original version of O++ is described in the document titled *Ode (Object Database and Environment): The Language and the Data Model*. Changes to O++ and current

O++ restrictions are described later. We assume that the reader is well versed in C++ [Stroustrup 1991].

An O++ database is identified by the name of the file in which the database resides. Multiple databases can be accessed from an O++ program but only one database can be accessed at any given time. An O++ program trying to create, access or otherwise manipulate database objects *must* first open (or create) a database. After manipulating objects in the database, it must be closed. Another one can then be opened.

When a simple name (with no embedded `'/'`) is used, O++ creates the databases in the database area specified by the environment variable `ODE_DEFAULT_AREA`. Thus, `ODE_DEFAULT_AREA` must be set to the name of a database area in which the databases to be created.

A database name can also be given as a full path name, i.e., starting with `'/'`:

full-pathname-of-database-area/database

8.1 O++ Compiler

The O++ compiler is called `oo`. It is used in a manner similar to the C++ compiler which is called `cc`. See the man page for details about `oo`.

8.2 Header File `ode.h`

Each O++ source file that uses O++ database facilities must include the file `ode.h`. If the O++ source file is just a C++ file (i.e., no O++ facilities are used), then this header file need not be included.

8.3 Class `database`

Class `database` (which is automatically made available by including the header file `ode.h`) provides functions for manipulating (closing, opening, etc.) the database and naming persistent objects. A ''truncated'' version of class *database*, which is automatically made available to an application as a result of including the header file `ode.h`, is shown below:

```
/*********************************************************************
A database is contained within a database area file or partition.
A default database area may be designated by an enviroment variable,
ODE_DEFAULT_AREA, and a database, db, within that area as
$ODE_DEFAULT_AREA/db.  If this variable is undefined,
followed by a ':' followed by a '/' as in host:/<path to area + database>).
Only one database can be open at a given time.
*********************************************************************/

…
class database {
    …
public:
    …
    static database* open(      // open (or create) a database
        const char *name,       // the name of the db
        int rdonly = 0,         // if true read only, else r/w
        int create = 1,         // if true and no such db exists, create db
        int trunc = 0           // if true clear the db
        );                      // it returns NULL on failure

    ERR   close(void);          // close this database
    ERR   remove(void);         // remove this database
    char* name(void);           // the full path name of this database

    static database& of(const persistent void * p); // db of ob pointed by p
    int    valid(const persistent void * p);      // valid pointer?

        // functions for the named object directory

    char* get_name(persistent void * id); // name of obj or NULL
    persistent void * & get_obj(char *name);      // null if no such name
    ERR   set_name(char *name, persistent void * id, int overwrite = 0);
              // give name to this object
    ERR   remove_name(persistent void * id);      // remove name of this obj

};
```

Only the following `database` operations

- `open`,
- `close`, and
- `remove`

can be invoked from outside a transaction body. All other operations must be invoked from within the transaction body.

8.3.1 `open`*:* The `static` function `open` opens the database identified by `name`. If the specified database is not found and the argument corresponding to the parameter `create` is specified to be non-zero, then the database is created with protection mode `mode`. Specifying the argument corresponding to the parameter `rdonly` to be 1 ensures that the database is opened in read-only mode. Currently, attempts to update the database will not be flagged as an error.

On successful completion, it returns the pointer to the *database* object. On failure, `open` returns `NULL`.

8.3.2 `close`*:* Member function `close` is used to close a database previously opened by a calling `database::open()`. `close` returns zero on success, non-zero on failure.

8.3.3 `remove`*:* Member function `remove` removes (deletes) a database previously opened by a calling `database::open()`. `remove` returns zero on success, non-zero on failure.

8.3.4 `name`*:* Member function `name` returns the full path name of the specified database.

8.3.5 `valid`*:* Member function `valid` returns a true or false depending upon whether or not its argument refers to a persistent object. In other words, `valid` returns false if the persistent pointer is a dangling pointer and false otherwise.

8.4 Transactions

All code interacting with the database (except database opening and closing), or manipulating objects must be within a transaction block. Ode uses 2-version 2-phase (2V2P) locking by (2-phase locking will be available as a compile time option) which simultaneously allows multiple transactions to read an object and one transaction to write an object.

Currently, there are three flavors of transactions: update, read only, and hypothetical. Update transactions have the form:

```
trans {  ...  }
```

Here is a code fragment illustrating the opening and closing of a database named `alex`:

```
#include <ode.h>
...
main()
{
    database *db;
    ...
    if ((db = database::open("alex")) == NULL) {
            error("cannot open database alex");
    }
    trans {
        ...
    }
    db->close();
}
```

The database must be closed and opened outside the transaction body.

Read-only transactions have the form

```
readonly trans {  ...  }
```

Persistent objects should not be updated in read-only transactions. Programs containing read-only transactions should be compiled with the `ODE_READONLY` constant defined using the `-D` option when

invoking the O++ compiler, e.g.,

```
OO -D ODE_READONLY …
```

All source files involving read-only transactions should be compiled with the above option, even though some or all of these files may contain update transactions. The update transactions will incur an extra check for each persistent object dereference.

If a persistent object is updated within a read-only transaction, then the semantics are undefined; in any case, the database will not be updated.

Read-only transactions are less likely to deadlock because they request only read locks and faster because no log records written.

Note that a read-only transaction can be run only if the data being read has been previously created. Otherwise, the transaction will attempt to update the catalog and thats not consistent with the read-only semantics.

Hypothetical transactions have the form

```
hypothetical trans {  …  }
```

Hypothetical transaction allow users to pose ‘‘what-if’’ scenarios (as often done with spread sheets). Users can change data and see the impact of these changes without changing the database. Only the transaction making the changes sees the changes because they are not made visible to other users. No log records are written.

8.5 Note on Locks

Write locks have more overhead than read locks. Unfortunately, the O++ compiler cannot always determine whether or not a persistent object is being accessed for reading only or for update. In such cases, the code generated assumes that the object will be updated which means that a write lock will be obtained. For example, the implicit coercion of a `persistent char *` (persistent string pointer) to a `char *` (transient string pointer) by passing it as an argument to a function expecting a `char *` argument will cause a write lock to be obtained on the persistent string. The `printf` function is an example of where this takes place. A read lock is obtained if the coercion is to a `const char *` as is the case when the C++ streams output operator `<<` is used to print a string since it expects a `const char *`.

8.6 Transaction Abortion

Transactions can be explicitly aborted by using the `tabort` statement which has the form

```
tabort;
```

In such a case, control simply flows to the statement, if any, following the transaction block.

The Ode system may also abort transactions to break deadlocks. In this case, the client application is aborted.

8.7 Persistent Objects

Objects of any class type and of the primitive types `char`, `short`, `int`, `long`, `float`, and `double` can be made persistent.

Definitions of classes whose objects are to be made persistent must be preceded by persistent forward declarations which have the form

`persistent class` *class-name*`;`

For example,

```
persistent class employee;
class employee {
    ...
};
```

If a class is to be used both in a C++ program and in an O++ program, then the preferred way of declaring a class as persistent is to give a persistent forward declaration. But if a class is to be only used in an O++ program, then instead of giving a forward declaration first and then giving its complete declaration, the class can alternatively be declared to be persistent directly:

```
persistent class employee {
    ...
};
```

Specifying a class *C* as persistent is not necessary in a file that contains code which does not manipulate or otherwise reference persistent objects of type *C*.

It is important that

1. a persistent forward declaration be included in all files that contain constructors for the type whose objects are or will be made persistent; and
2. persistent forward declarations also be given for all classes involved in the definition of a persistent class. (One exception to this is a class used to declare volatile pointers within a persistent class).

8.8 Creation and Deletion of Persistent Objects

Persistent objects are created and deleted using the operators `pnew` and `pdelete`. Except for the fact that these operators allocate objects in the database, their semantics are similar to those of the operators `new` and `delete`. Operators `pnew` and `pdelete` cannot be overloaded.

Operator `pnew` returns a pointer to persistent object. Such pointers are referred to as *persistent* pointers (object ids). Persistent pointers are declared like ordinary pointers except that the type qualifier `persistent` is used. Note that the value `0` and the constant `NULL` are automatically coerced to null persistent pointers.

Persistent objects are referenced using persistent pointers much like heap objects are referenced using volatile pointers.

Here is some example code illustrating the creation and manipulation of persistent objects:

```
persistent class employee;
class employee {
    ...
public:
    ...
    char name[MAX];
    int age;
};
    ...
persistent employee *pe;
    ...
pe = pnew employee;
    ...
pe->age = 40;
```

‘‘Volatile’’ pointers embedded within persistent objects will not have legitimate values across transactions. For example, had `name` been implemented as a pointer to a dynamically allocated array (i.e., of type `char *`), then it would not have a valid value across transactions.

8.9 Queries

Objects of class types (only!) can be accessed by using the associative `for` loop as illustrated:

```
for (pe in employee)
    cout << "Name = " << pe->name << ", Age = " < pe->age << endl;
```

This `for` loop will ‘‘iterate’’ over all the `employee`s in the database, i.e., over the `employee` ‘‘type extent’’. After the completion of the loop, `pe` will have the value `NULL`.

If we are only interested in subset of objects, then we can use the `suchthat` clause to restrict the set of objects examined. For example, we can look at only employees older than 25 years as follows:

```
for (pe in employee) suchthat(pe->age > 25)
    cout << "Name = " << pe->name << ", Age = " < pe->age << endl;
```

We can access a specific `employee` object as follows:

```
for (pe in employee) suchthat(strcmp(pe->name, "O. Shmueli") == 0)
    cout << "Name = " << pe->name << ", Age = " < pe->age << endl;
```

Joins can be performed using nested `for` loops or a loop with multiple loop variables. For example,

```
for (pe in employee; pd in department) {
    ...
}
```

8.10 Named Persistent Objects

Persistent objects can be named. The names enable fast access of these objects; for such objects, it is not necessary to use the general `for` query statement to access them. For example, assume that `db` represents an open database. The persistent `employee` referenced by `pe` whose `name` field has the value `"O. Shmueli"` can be named `oded` by using the function `set_name` as follows:

```
db->set_name("oded", pe);
```

`db->set_name` returns zero if it is successful and a non zero value in case of an error.

In another transaction, this object can be retrieved by using `get_obj` as follows:

```
pe = (persistent employee *) db->get_obj("oded");
```

`get_obj` returns zero (the null persistent pointer) in case of an error. It is the user’s responsibility to insure that the type of named object is the one to which it is cast.

The above object could also have been retrieved by using the `for` loop as

```
...
for (pe in employee) suchthat(strcmp(pe->name, "O. Shmueli") == 0) {
    ...
}
```

but this may be slower.

The member function `get_name` of class `database` is used to retrieve the name of an object:

```
char *name, *p;
...
if (p = db->get_name(pe)) {
    name = new char [strlen(p)+1];
    strcpy(name, p);
}
```

If no name is associated with an object, then `get_name` returns `NULL`.

The relationship between names and objects is one-to-one; i.e., an object may have at most one name, and a name may correspond to at most one object. The persistent object naming mechanism is intended for providing fast access to objects that have been stored in the database under a well known name.

8.11 Persistent Arrays

O++ allows the user to allocate and access persistent arrays, in a manner analogous to C++. A persistent array is allocated (dynamically) by specifying it's size. As in C++, a default constructor must exist for the type, and it will be used to initialize each object in the array.

```
persistent employee *pe;
int size;
    ...
pe = pnew employee [size];
```

Individual objects are accessed using the subscript operator []:

```
pe[3].age = 40;
```

8.12 Pointer Arithmetic

Pointer arithmetic can be performed with pointers to persistent objects much like with ordinary pointers.

8.13 Casting Persistent Pointers to volatile Pointers

Persistent pointers can be cast to volatile pointers. The volatile pointer refers to the memory address of the persistent object. Ode will bring the object from disk to the buffer pool if the object is not already in memory.

8.14 Large Objects

An object larger than a ''page'' is classified as large. The current page size is 4K bytes.

8.14.1 Header File `large.h`*:* Large objects are, by default, handled transparently. However, applications may find it more efficient to manipulate large objects by explicitly accessing portions of such objects. Class `large` provides functions for efficiently manipulating large objects. Each O++ source file that uses special large object interface provided by the class `large` must include the file `large.h`. A ''truncated'' version of class `large`, which is defined in the O++ header file `large.h`, is shown below:

```
/*****************************************************************************
     A byte range is defined as a (b, n) pair,  where  b  is  the
     start byte and n the number of bytes of the byte range; i.e.,
     byte range (b, n) includes the region of bytes from b up  to
     and including byte b+n-1.
*****************************************************************************/
…
class large : private _handle {
public:
    large();
    ~large(void);

    ERR create(int n, char *buf = NULL, int hint=0, const _cluster* c = NULL);
    ERR open(const persistent void * p);
                // open this object for manipulation as large object
    ERR close(); // close the object

    ERR read(char **p, int b, int n);   // b can be eliminated if cursor
    ERR write(char *p, int b, int n);
    ERR append(char *p, int n, int hint=0);
    ERR insert(char *p, int b, int n);
    ERR remove(int b, int n);
    ERR truncate(int b);

    ERR read(char *buf, int b, int n); // read (b,n) at this specific buf

    ERR getf(FILE* fp, int n);
        // get n bytes from fp starting from the current R/W position
        // and append them to (possibly 0 length) lo;
        // after this operation R/W position += n

    ERR getf(FILE* fp);
        // get all bytes from fp starting from the current R/W position
        // and append them to (possibly 0 length) lo;
        // after this operation R/W position is at the end of fp

    ERR putf(FILE* fp, int b, int n);
        // get (b,n) bytes from lo and write them in fp starting from the
        // current R/W position; after this operation R/W position += n

    ERR putf(FILE* fp, int b = 0);
        // get (b, n) bytes of lo (where n is all bytes on the right of b) and
        // write them in fp starting from the current R/W position;
        // after this operation R/W position += n

    int size();         // size of this
    int thresh(int t);  // set threshold to t, returns the current threshold
    int level();        // level of the tree pointing to segments
    int npages();       // number of pages the object is stored

    ERR pack(); // store the object in as few and large segms as possible
    ERR flat(); // make the directory flat (level 1) if it is not

    ERR cat(const large& lo, int b, int n, int hint);
                // append to this the (b, n) bytes of lo
};
```

For examples illustrating the use of class `large`, see the example programs in files `get.c` and `store.c` in the directory *ode_directory*`/o++2.0/src/demo`.

8.15 Versions

Object versioning in Ode is orthogonal to type, that is, versioning is an object property and not a type property. Versions of an object can be created without requiring any change in the corresponding object type definition, all objects can be versioned, and different objects of the same type can have a different number of versions. O++ tracks both temporal and derived-from relationships.

Suppose `p`, `v1`, `v2`, and `v3` are declared as

```
persistent employee *p, *v1, *v2, *v3;
```

Assume that `p` refers to an `employee` object. Then a new version of this object can be created and its id stored in `v1` as follows:

```
v1 = newvers(p);
```

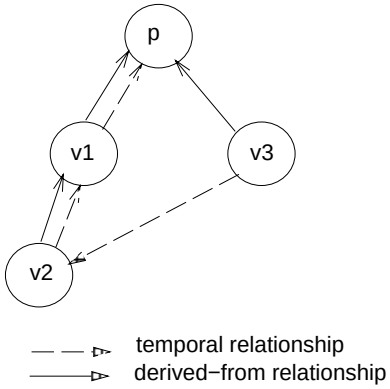
Ode records the fact that the object referenced by `v1` was ‘‘derived’’ from the object referenced by `p`. Function `vdprev` (`v`ersion `d`erived-from `prev`ious) can be used to extract this information. The call `vdprev(v1)` will yield `p` and the call `vdprev(p)` will yield `NULL` (assuming that `p` is not a version of another `employee` object).

Ode also records that the fact that object referenced by `v1` was created after that referenced by `p` (assuming that no other versions of this `employee` object were created in the meantime). Function `vtprev` (`v`ersion `t`emporally `prev`ious) can be used to extract this information. The call `vtprev(v1)` will yield `p` and the call `vtprev(p)` will yield `NULL`.

To further illustrate the information recorded by Ode, suppose that the following statements are now executed:

```
v2 = newvers(v1);
v3 = newvers(p);
```

The versions created above are related as follows:



Calling `vdprev(v2)` will yield `v1`, `vdprev(v3)` will yield `p`, `vtprev(v2)` will yield `v1`, and `vtprev(v3)` will yield `v2`.

Function `vlatest` (`v`ersion `latest`) is used for finding the most recently derived version of a particular object. For instance, `vlatest(p)` will yield `v3`, `vlatest(v1)` will yield `v2`, `vlatest(v2)` will yield `v2`, and `vlatest(v3)` will yield `v3`.

Function `vroot` (`v`ersion `root`) is used to find the initial object that an object is directly or indirectly derived from (the farthest back one can go by applying `vdprev` to a version repeatedly before hitting `NULL`). For our example, `vroot(p)`, `vroot(v1)`, `vroot(v2)`, and `vroot(v3)` all evaluate to `p`.

All the objects derived from some initial object are in some sense related. The most recently created version in this relation can be computed given any version `v` using `vlatest(vroot(v))`. Given this latest version, all related objects can be found by following the `vtprev` links back to the root.

For examples illustrating the use of class `large`, see the example programs in files `vers.c`, `versEmp.c` and `listvers.c` in the directory *ode_directory*`/o++2.0/src/demo`.

9. Specialized Facilities for Advanced Users

The specialized facilities are automatically made available to users by including the header file `ode.h`

9.1 Flexible Transaction Begin and End

Instead of using transaction blocks, users can begin a transaction by calling

```
_transaction::begin()
```

commit a transaction by calling

`_transaction::commit()`

and abort a transaction by calling

`_transaction::abort()`

Using these member functions means that O++ will not be able to detect nested transaction calls (which are not supported), jumping in and out of transaction code, and other potential errors. Also, some facilities such as those for versioning will not be automatically available. In case of `trans` blocks, the O++ compiler generates code to initialize and close versioning facilities at the beginning and end of a transaction.

9.2 Scanning a Type Extent

A ‘‘type extent’’ refers to the collection of all objects of the same type in the database. Users can examine the objects in a type extent (class types only, no primitive types or arrays!) by using the O++ `for` statement. Some applications, such as a database browser, may want low-level control for examining the objects in the database. For example, such an application may want to look at the first object in the database, examine the next object, or examine the previous object.

A ‘‘cluster’’ is a physical concept that refers to where groups of objects are stored. By default, all objects of a type *t* are stored in the cluster named `_cluster_`*t*. For example, all `employee` are stored in the cluster `_cluster_employee`.

Class `scan` contains facilities for explicitly scanning the objects in a cluster; these facilities are automatically made available to an application as a result of including the header file `ode.h`:

```
/*********************************************************************

Facilitates cluster- and page-oriented  forward and  backward
scanning of objects in a given cluster. The state of a scan is recorded
by a cursor. The cursor points to  the "current" object in the cluster
being scanned.  Scans are independent from each other even if they
refer to the same cluster so that nested loops on the same cluster
can be supported.

The 'inpage' parameter of the scan functions control wether the scan is
file-oriented (inpage is false) or page-oriented (inpage is true).
In the latter case, only objects in the current page of the cluster
being scanned are returned. To proceed to an other page within
the cluster, the 'inpage' value must be set to false.

When scanning a cluster, only first-level ordinary objects are included
in the scan.
*********************************************************************/

class scan {
   …
public:
    scan(const _cluster& c);        // make a scan for cluster
    scan(const database& db);       // make a scan for db
    ~scan();

    ERR first(int inpage = 0);
    ERR last(int inpage = 0);
    ERR next(int inpage = 0);
    ERR prev(int inpage = 0);

    ERR at(const persistent void * & oid); // seek at this object; oid must
                                    // be a member of the cluster
                                    // being scanned.

    persistent void * current(void);
                    // if the cursor currently points to an object
                    // within the cluster it returns the oid of this
                    // object otherwise, it returns _OID_NULL.
};
```

File `escan.c` in the directory *ode_directory*`/o++2.0/src/demo` contains an example illustrating building a trivial browser by scanning a type extent.

9.3 Catalog

The catalog consists of elements of the predefined type `metatype`, which is automatically made available as a result of including file `ode.h`. The types stored in the database can be examined by iterating over the `metatype` type extent. Class `metatype` is defined as

```
//
// metatype describes all types in a database
//
class metatype {
public:
    char name[MAX_TYPE_NAME];   // type name
    unsigned int size;          // size of instance of this

    int basecnt;        // how many base classes
    _PERS_PTR(base_spec) base_list;     // list of base classes
};


class base_spec {       // persistent class descriptor
public:
    Access access;         // one of private/public/protected
    boolean is_virtual;     // is it a virtual function?
    _PERS_PTR(metatype) baseclass;      // persistent descriptor of base class
    base_spec() {};
    base_spec(Access a, boolean iv, _PERS_PTR(metatype) p)
        { access = a; is_virtual = iv; baseclass = p; }
};
```

Note that the definition of class `metatype` is likely to change. File `catalog.c` in the directory *ode_directory*`/o++2.0/src/demo` contains an example illustrating how information in the catalog can be accessed.

10. Miscellaneous Facilities

The O++ compiler OO defines the constant `__oplusplus` to allow users to determine whether or not the program is being compiled with the O++ compiler.

11. Examples

For examples of O++ programs, see the directory

ode-directory-path`/o++2.0/bin/src/demo`

12. Deadlocks

Deadlocks can result when multiple transactions access the same objects concurrently. In such cases, the Ode server will abort one or more of transactions involved in the deadlock to break the deadlock. Currently, abortion of a transaction by the Ode server results in the termination of the client process containing the transaction.

13. Recovery

The Ode server (EOS) creates log files (in the file specified in the initialization file `~/.eos/serverrc`).

13.1 System Failure

After the occurrence of a system failure, restarting the Ode server will *automatically* return the database to the last consistent state prior to failure, as recorded in the log. The log is scanned and updates made by the committed transactions are redone in exactly the same order as they were originally performed. If a system failure occurs during the restart, the subsequent restart performs the same work again in an idempotent fashion.

13.2 Restarting the Server

If the server crashes then the following things might happen:

1. Shared memory segments remain in the system.
2. Semaphores remain in the system.
3. Message queues remain in the system
4. The disk daemon process `eos_diskd` was not notified and is therefore still running.
5. The log process `eosserver` (same name as the server) was not killed.

Presence of shared memory segments, semaphores, and message queues, which are inter-process communication (IPC) facilities, can be detected by using the UNIX command `ipcs`. If any IPC facilities used by the crashed server process have not been freed, then these should be removed (freed) by using the UNIX command `ipcrm` before restarting the server.

If the disk daemon process `eos_diskd` and the log process `eosserver` are around, (check by executing the command `ps -aux`), then they should be killed with the `kill` command.

14. Protection Against Media Failure

Ode does not currently provide automated facilities to protect against media failure. User can protect themselves against media failure by explicitly making a backup copy of a database area (which may contain one or more databases) as follows:

1. Make sure that no active transaction exists in the system.
2. Checkpoint the database by giving the server the command

   ```
   checkpoint
   ```

   Shutdown the server with the command

   ```
   shutdown
   ```

3. Copy the database area to another disk or to any other place.

A database area is restored from a backup copy as follows:

1. Remove the global log and checkpoint files (their path names are specified in the initialization file `serverrc` — the values of the variables `EOS_CHECKPOINT_NAME` and `EOS_GLOBAL_LOG_NAME`).
2. Remove all the private log records generated; these are files beginning with the prefix `priv_` (the private log records are in the directory specified in the initialization file `serverrc` — the value of the variable `EOS_PRIVATE_LOG_DIR`). These records are not needed any more, so they are removed to free space.
3. Copy the backup copy of the database to the directory where the database was originally. was before
4. Start the server.

15. O++ Language Changes

The original version of O++ is described in the document titled *Ode (Object Database and Environment): The Language and the Data Model*. This section describes the changes to O++ and implementation restrictions.

15.1 Additions

1. Large objects can be manipulated in a file-like fashion using the class `large`.
2. Users can access information in the catalog which consists of persistent objects of type `metatype`.
3. Objects in a type extent can be scanned using the facilities provided by class `scan`. Unlike the `for` loop, these facilities can be used to examine objects in an application specific order; scanning can be

suspended and resumed arbitrarily.

4. Databases must be explicitly opened and closed. Users can access multiple databases from within a single program but only one database can be accessed at a time.
5. Three flavors of transactions have been added:
    1. Update transactions are written as

    ```
    trans {
        ...
    }
    ```

    2. Hypothetical transactions are written as

    ```
    hypothetical trans {
        ...
    }
    ```

    3. Read-only transactions are written as

    ```
    readonly trans {
        ...
    }
    ```

6. Named persistent objects.
7. Objects of primitive types can now be made persistent.
8. Pointer arithmetic allowed on persistent objects.

15.2 Restrictions

1. Persistent pointer types cannot be used to differentiate between signatures of overloaded functions. For example,

    ```
    int insert(persistent employee *pe);
    int insert(persistent item *pi);
    ```

    the code produced by O++ will be flagged by the C++ compiler as an error because each persistent pointer is translated to an object of type `_pref`.

2. Persistent pointer parameters cannot be given default values because the generated C++ code invokes a constructor to perform the initialization; constructor calls for default parameter values are not currently implemented in C++.
3. `pdelete` will call the destructor, if any, of the object type as determined from its argument (a persistent pointer). If the pointer is of a base class and the object of a derived class, then the destructor that will be invoked will be of the base class and *not* the derived class. Thus virtual destructors are not of help when using `pdelete`.

    If the use of virtual destructors is essential, then `pdelete` should be packaged within a virtual functions to and these functions used to delete objects.

15.3 Not Implemented

1. Sets.
2. `forall` loops.
3. Triggers.
4. Constraints.
5. Subclusters.

6. Dual Pointers.

15.4 Syntax & Semantic Modification

1. A `continue` inside a `for` loop with multiple iterators gets the next iteration (combination of values).
2. Static members: the value of a static data member of a class is the same for all persistent instances created on the same invocation of the program.
3. The `is` operator is now `=>`. The syntax is now

   persistent-pointer `=>` *object-class*

   Operator `=>` returns true if *persistent-pointer* refers to an object of type *object-class*.

15.5 Beware!

1. It is necessary to open a database to scan objects and create new objects. However, an object can be accessed or deleted, assuming its oid is known, without opening a database. Users should be careful when writing applications that manipulate multiple databases.
2. A database can be opened in read-only mode; currently, attempts to update the database will not be flagged as an error.
3. As in the case of volatile objects, the special variable `this` is a volatile pointer for persistent objects. It is not a pointer to the persistent object; instead it points to the in-memory copy of the persistent object. However, within a member function, the special variable `pthis` can be used much like the special variable `this`. Provided the member function has been called in conjunction with a persistent object, `pthis` will yield a persistent pointer of the right type. Otherwise, it will yield the null value.
4. `pthis` may not work correctly if `this` points to a base class object in the presence of multiple inheritance.
5. `static` members of a class are shared between class objects. Such components are not made persistent (because C++ does not implement them as belonging to the object).
6. Members of persistent objects that are pointers to volatile objects will not in general have meaningful values across transactions.
7. Applications accessing the catalog are likely to be affected by future releases of Ode since the definition of class `metatype` is likely to change.
8. Modifying the schema (class definitions) corresponding to existing objects will lead to unspecified behavior such as core dumps.

15.6 Notes

1. The SunOS must be configured to support shared memory, semaphores, and message queues. Otherwise, the server will not be able to start. The system should be configured (reboot required) to support shared memory, semaphores, and message queues.
2. Currently, a single database can be at most 2.1 gigabytes in size.
3. Persistent pointers cannot be assigned to volatile pointers without explicit casting.
4. Persistent objects cannot be passed by reference as illustrated below:

```
persistent class employee;
class employee { … };

void f(employee& a) { … }

int main()
{
    employee *e;
    persistent employee *pe;
    …
    trans {
        …
        f(*e); // allowed
        f(*pe); // not allowed
    }
}
```

If passing persistent objects was legal, then parameter `a` would refer either to objects on disk or in memory depending upon what was passed. Incidentally, the current error message says that you cannot assign a pointer to a persistent object to an ordinary pointer reflecting the implementation of passing by reference in C++.

5. Persistent pointers cannot be members of union. O++ translates a persistent pointer to a class (with constructors) object, and C++ does not allow such members.

15.7 O++ Keywords

```
by       constraint deactivate dual
forall   in         is         hypothetical
pdelete  perpetual  persistent pthis
pnew     readonly   suchthat   tabort
trans    trigger    vlatest    newvers
vdprev   vtprev
```

16. More Documentation

The *Ode: Object Database & Environment* is a collection of reports describing the Ode database, its design, and other related topics. If you would like a copy, please send mail to

`ode@allegra.att.com`

along with your mailing address.

17. Help and Bug Reports

To get help with installation or using O++ and to report (potential) bugs send mail to

`ode@allegra.att.com`

When reporting (potential) bugs or problems, please send a complete source file causing the (potential) bug or problem. It is important that this file be complete and as small as possible with extraneous stuff deleted.

18. Additional Interfaces to be Provided in Future Releases

Besides O++, several other Ode interfaces are under development. These interfaces, which will be available in later releases of Ode, are

1. OdeView: A graphical X-window based interface to Ode
2. CQL++: An SQL-like interface to Ode

3. OdeFS: A UNIX system file-like interface to Ode

19. Ode Publications (Partial List)

1. Agrawal, R. and N. Gehani 1989. Ode: Object Database & Environment. *SIGMOD*, Portland, Oregon.
2. Agrawal, R. and N. Gehani 1989. Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++ (Expanded Version with R. Agrawal). *2nd International Workshop on Database Programming Languages*, Portland, Oregon.
3. Agrawal, R., S. Buroff, N. Gehani, and D. Shasha. Object Versioning in Ode. *1991 IEEE Data Engineering Conference*, Kobe, Japan.
4. Agrawal, R., N. Gehani, and J. Srinivasan 1990. OdeView: The Graphical Interface to Ode. *SIGMOD*, Atlantic City, 1990.
5. Biliris, A. 1992. An Efficient Database Storage Structure for Large Dynamic Objects, *IEEE Data Engineering Conference*, Phoenix, Arizona. February 1992
6. Biliris, A. 1992. The performance of Three Database Storage Structures for Managing Large Objects, *ACM-SIGMOD Conference*, San Diego, California. June 1992
7. Biliris, A. and E. Panagos 1993. EOS User's Guide. AT&T Bell Laboratories, Murray Hill, NJ 07974.
8. Dar, S., N. Gehani and H. V. Jagadish 1992. CQL++: a SQL for a C++ based Object-Oriented Database. *Proc. of Int'l Conf. on Extending Database Technology*, Vienna, Austria.
9. Biliris, A., S. Dar, and N. Gehani. Making C++ Objects Persistent: Hidden Pointers. To be published in *Software — Practice & Experience*.
10. Gehani, N., H. V. Jagadish and W. D. Roome 1993. OdeFS: A File System Interface to an Object-Oriented Database AT&T Bell Laboratories, Murray Hill, NJ 07974.

Ode 2.0 User's Manual

A. Biliris
N. Gehani
D. Lieuwen
E. Panagos
T. Roycraft

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Ode is a database system and environment based on the object paradigm. It offers one integrated data model for both database and general purpose manipulation. The Ode 2.0 database is defined, queried and manipulated using the database programming language O++, an extension of C++. O++ provides facilities for creating persistent objects, defining sets, and queries.

This document describes

1. how to install Ode,
2. how to start the storage manager,
3. a brief summary of O++,
4. using O++,
5. changes to O++, and
6. other items of interest.

It also contains a list of all the Ode publications.