The EYEDB OODBMS

Eric Viara
Sysra Informatique
7, rue de Bièvres
92140 Clamart, France
viara@sysra.com

Emmanuel Barillot
GIS Infobiogen
7, rue Guy Môquet – BP 8
94801 Villejuif cedex, France
and
Généthon
1bis, rue de l'Internationale
91000 Évry, France
emmanuel.barillot@infobiogen.fr

Guy Vaysseix
GIS Infobiogen
and
Généthon
vaysseix@infobiogen.fr

Abstract

This paper introduces the EYEDB *Object Oriented DataBase Management System (OODBMS).* EYEDB *implements all the standard features of OODBMS, is language oriented, provides a generic object model and a support for data distribution using CORBA. It can deal with very large databases, and is both efficient and scalable. It is used in the genome project where huge amount of data have to be managed and intricate data structures needs to be modeled. Online information and a trial version of* EYEDB *can be obtained from http://www.sysra.com/eyedb.*

1. Introduction

The development of EYEDB was initiated in 1993 at Généthon to store and facilitate access to the human genome mapping data (physical and genetic maps). At that time, EYEDB was intended as a persistent OO system, that was very light and much more efficient for storage and retrieval than the other OODBMS. From 1994, Sysra Informatique has rewritten completely EYEDB and made it a real OODBMS.

The key features of the EYEDB OODBMS are:

- **standard OODBMS features** [4, 18, 14]: persistent typed data management; client/server model; transactional services; recovery system; expressive object model; inheritance; integrity constraints; methods; triggers; query language; application programming interfaces,
- **language orientation**: a definition language based on the ODMG [11] Object Definition Language (ODL); a query language based on the ODMG Object Query Language (OQL); C++ and Java bindings,
- **genericity and orthogonality of the object model**: inspired by the SmallTalk, LOOPS, Java and ObjVlisp object models (i.e. every class derives from the class `object` and can be manipulated as an object); type polymorphism; binary relationships; literal and object types; transient and persistent objects; method and trigger overloading; template-based collections (set, bag and array); multi-dimensional and variable size dimensional arrays,
- **support for data distribution**: CORBA binding; multi-database objects,
- **support for large databases**: databases up to several Tb (tera-bytes),
- **efficiency**: database objects are directly mapped within the virtual memory space; object memory copy are reduced to the minimum; clever caching policies are implemented,
- **scalability**: programs are able to deal with hundred of millions of objects without loss of performance.

The system architecture is briefly exposed in section 2. Section 3 introduces the storage manager subsystem. In section 4, the object model is exposed. Sections 5 and 6 describe the EYEDB object definition language and query language. Sections 7, 8 and 9 deal with the C++, Java and Corba bindings. In section 10, we present briefly some applications

using EYEDB in the biological domain. A few comparison elements between EYEDB and the other OODBMS are exposed in section 11.

2. The Architecture

EYEDB is based on a client/server architecture as shown in Figure 1.

The EYEDB server is composed of:

- the server protocol layer based on Remote Procedure Call (RPC),
- the object model implementation,
- the OQL engine,
- the storage manager subsystem.

A client is composed of:

- the user application code,
- the C++ (resp. Java) API implementing the C++ (resp. Java) binding,
- the client protocol layer based on RPC.

3. The Storage Manager Subsystem

The storage manager subsystem provides the following main services:

- persistent raw data management,
- transactional services,
- recovery system,
- B-tree and hash indexes,
- multi-volume database management.

The storage manager can be used independently from EYEDB.

3.1. Persistent Raw Data Management

The central concept of the storage manager is the raw object. A raw object is a piece of persistent raw data tied to an object identifier named *oid* .
An *oid* identifies a raw object in a unique way within a set of databases. It is generated by the storage manager at raw object creation.
An *oid* is composed of three fields: the storage index, the database identifier and a random generated magic number.
The first field identifies the object physical location within a database volume. The second one identifies a database and the last one ensures more security in the object identification process.

The storage manager is responsible for the management of raw objects:

- raw object creation (*storage allocation, oid allocation, object storage*),
- raw object update (*content modification*),
- raw object reading,
- raw object deleting (*oid deallocation, storage deallocation*),
- raw object resizing (*storage reallocation, object moving*),
- raw object locking and unlocking (*shared locking, exclusive locking, private locking*),
- raw object access control

3.2. Memory Mapped Architecture

The storage manager is based on a memory mapped architecture. Database volumes are mapped within the server virtual memory space.
Due to some 32-bit system limitations, the databases greater than 2Gb cannot be mapped as a whole.

The storage manager implements a segment-based mapping algorithm: when reading an object, the storage manager checks if the corresponding storage piece is mapped within its virtual memory space. If it is not mapped, it maps a large segment of data around the raw object, eventually extending a neighboring segment.
If the total mapped size is more than the system maximum, it unmaps the less recently used mapped segment. On 64-bit system, this algorithm is not needed as databases up to several Tb (i.e. tera-bytes) can be mapped at the whole within the virtual memory space.
Currently, the storage manager can deal with databases up to one Tb.

3.3. Transactional Services

The storage manager provides standard transaction services which guarantees atomicity, consistency, isolation and integrity within a database.
Its transaction unit is based on a two-phase locking protocol. The protocol requires that each transaction issues lock and unlock requests in two phases:

- growing phase: a transaction may obtain locks but may not release any lock.

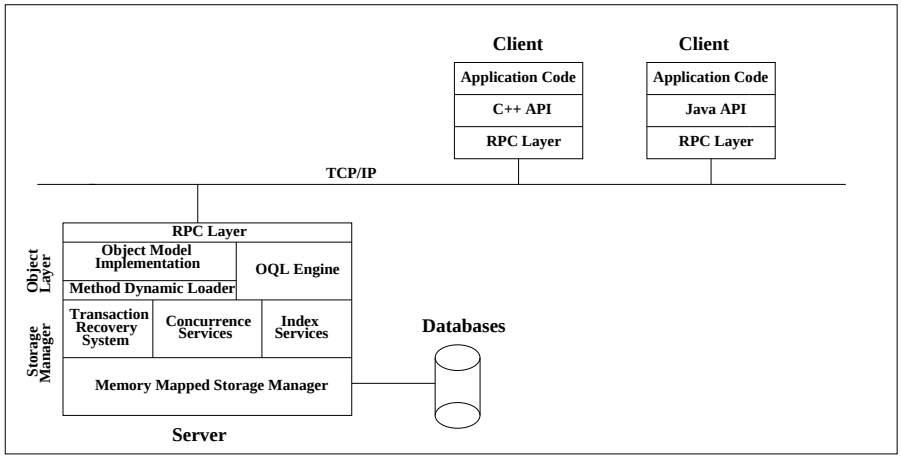

Figure 1. The EYEDB Architecture

- shrinking phase: a transaction may release locks but may not obtain any new lock.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase and no more lock requests may be issued.

The storage manager provides different transaction locking modes: read and write shared, read shared and write exclusive, read and write exclusive or database exclusive.
It also provides immediate deadlock detection.

3.4. The Recovery System

The storage manager provides a simple but efficient recovery system against failures:

- client failure: the transaction is automatically aborted by the server.
- server failure or operating system failure: the current transactions will be automatically aborted on the next database opening.
- the disk failure recovery is not supported: this is a deliberate choice of simplicity since storage consistency can rely on the RAID technology or transactional file systems now available on modern operating systems.

3.5. B-Tree and Hash Indexes

The storage manager provides support for B-Tree and Hash indexes.
The B-Tree index provides fixed size raw data indexation, efficient exact match query and range query.

The Hash index provides variable size raw data indexation and efficient exact match query. The hash key function can be provided by the client.

3.6. Multi-Volume Database Management

The database storage unit is the volume file. A database can contains up to 512 volumes each one up to 2Gb on a 32-bit file system interface, or up to several tera-bytes on a 64-bit file system interface. The storage manager provides facilities to add, move, resize and reorganize database volumes.

4. The Object Model

The EYEDB object model is inspired by the SmallTalk, LOOPS, ObjVlisp, Java and ODMG [11] models.

The main three class abstractions are the class `object` which is the root class, the class `class` and the class `instance` as shown in Figure 2.

An instance cannot be instantiated except the instances of the class `class` or the instances of the classes which inherit from the class `class` : the instantiation of an instance of the class `class` is an instance of the class `instance` .

If *new()* denotes the instantiation method:

```
struct _class Person =
    struct _class ->new(name = "Person", ...)
```

`Person` is an instance of the class `struct _class` . It is also a class that inherits from the class `struct` .

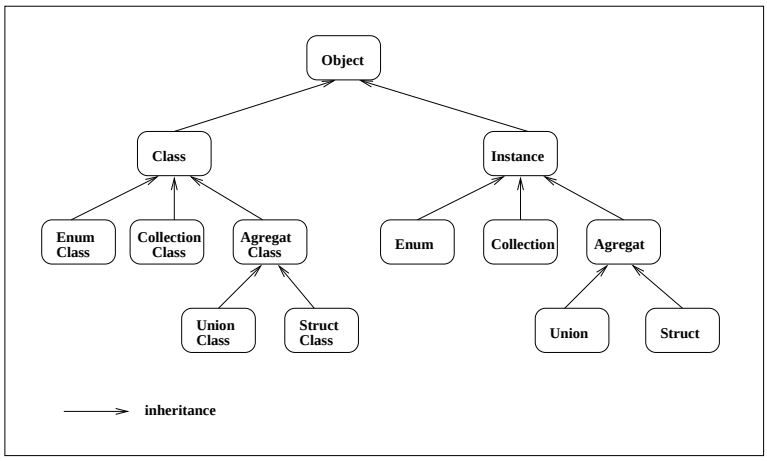

Figure 2. Partial Native Object Model

Person can be instantiated as follows:

```
struct john =
    Person->new(name = "john", age = 32)
```

john is an instance of the class struct that cannot be instantiated

To create a class Employee which inherits from the class Person :

```
struct _class Employee =
    struct _class ->new(name = "Employee",
    parent = Person, ...)
```

Employee is instantiable as follows:

```
struct henry =
    Employee->new(name = "henry",
    salary = 10000)
```

Figure 3 shows these instantiation mechanisms.

Note that as the class class derives from the class object , an instance of the class class can be manipulated like any instance of the class object .

The native EYEDB object model is composed of 76 classes such as the class collection , the class method , the class constraint , the class index , the class image and so on.
EYEDB object model supports all standard built-in types: 16-bit, 32-bit and 64-bit integer, character, string, 64-bit float .

An instance can be *transient* or *persistent* :

- an instance is *transient* if its lifetime does not exceed the lifetime of the unit of execution in which it is manipulated,
- otherwise the instance is *persistent* .

A *persistent* instance can be *object* or *literal* :

- an *object persistent* instance has an unique identifier (i.e. an *oid*),
- a *literal persistent* instance has no identifier.

4.1. Class Structure

A class is composed of a name, a parent class (except for the class object which is the root class), a set of attributes, a set of methods and a set of triggers:

- an attribute is composed of a type, an optionnal array modifier and is *literal* or *object* . For instance, using the EYEDB ODL language:

  ```
  attribute int32 age
  ```

  is a *literal* attribute of type int32 with no array modifier, while the following attribute:

  ```
  attribute Person *children[10]
  ```

  is a fixed-size array of *object* of type Person .

- a method is a unit of execution tied to a class.
  A method can be either a class method or an instance method.

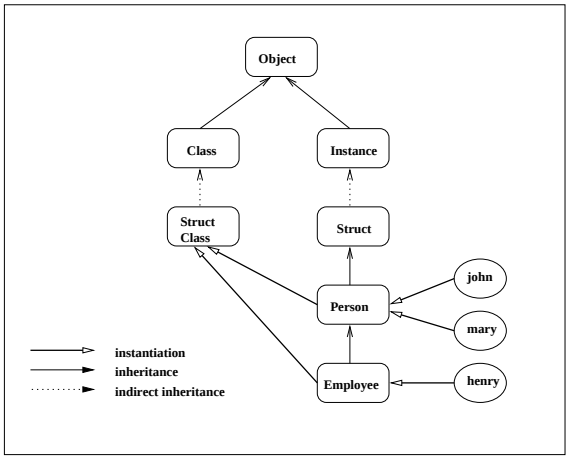

Figure 3. Applicative Object Model Example

- a trigger is a unit of execution tied to a class. Triggers are applied to instances of this class on a given event. For example, a trigger `update_before` tied to the class `X` means that before the update of any instance of the class `X`, the trigger will be called.
  A method or a trigger can be overloaded by the sub-classes.

  EYEDB supports the following trigger events: `create_before`, `create_after`, `update_before`, `update_after`, `load_before`, `load_after`, `remove_before`, `remove_after`.

4.2. Type Polymorphism

The two language bindings, C++ and Java, and EYEDB OQL supports type polymorphism: variables may be bound by instances of different types.
This is a direct consequence owing to the fact that any EYEDB class inherits from the class `object`,

The possibility of manipulating polymorphic objects is a major contribution of object orientation.

4.3. The Collection Type

A collection is composed of elements of the same type. The elements can be either *`literal`* or *`object`*.

If the collection element type is the class `object`, then the collection can contain instances of any class, as all classes inherit from the class `object`.

The collection types supported by EYEDB are the `set`, the `bag` and the `array`:

- an instance of the class `set` is an unordered collection with no duplicates allowed,
- an instance of the class `bag` is an unordered collection that may contain duplicates,
- an instance of the class `array` is a dynamically sized ordered collection.

The collection type is a major concept of the EYEDB object model.

4.4. Relationships

The EYEDB object model supports only binary relationships, i.e. relationships between two types.
A binary relationship may be one-to-one, one-to-many or many-to-many depending on the cardinality of the related types. Relationships are not named.

EYEDB maintains the referential integrity of relationships. This means that if an object that participates in a relationship is removed, then any traversal path to that object is also removed.
EYEDB supports object-valued attribute: this kind of attribute enables one object to reference another without expectation of referential integrity. An object-valued attribute implements a unidirectionnal relationship: in this case, EYEDB does not guarantee the referential integrity. Note that such a unidirectionnal relationship is not called a relationship.

The example introduced in the section *The Object Definition Language* illustrates the use of relationships and object-valued attributes.

4.5. Constraints

EYEDB supports all standard constraints:

- the `not null` constraint on a attribute within a class `X` means that no instances of the class `X` can have this attribute value not assigned.
- the `unique` constraint on a attribute within a class `X` means that one cannot create an instance of the class `X` which has the same attribute value than an existing instance in the database.
- the `cardinality` constraint on an instance of the class `collection` means that the count of this collection must follow this cardinality constraint.

5. The Object Definition Language

The EYEDB Object Definition Language (ODL) is a language based on the ODMG ODL to define the specifications of object types.
ODL is not intended as a full programming language, it is a definition language for objet specifications.

Like ODMG ODL, EYEDB ODL defines classes (inheritance and attributes), relationships and method signatures. EYEDB ODL extends the ODMG ODL to allow for the definition of attribute constraints (notnull, unique, collection cardinality), index specifications and trigger declarations. Unlike ODMG ODL, any instance of a class can be used either as a *literal* or as an *object*. EYEDB ODL also allows the user to specify whether a method is backend (i.e. server side) or frontend (i.e. client side), and whether it is a class or instance method.

Here is a simple example of an EYEDB ODL construct:

```
enum CivilState {
  Lady = 0x10,
  Sir = 0x20,
  Miss = 0x40
};

class Address {
  attribute char street[];
  attribute char town[32];
};

class Person {
  attribute char name[] (index[]);
  attribute int age;
  attribute Address addr;
  attribute CivilState cstate;
  attribute Person * spouse
            (inverse<Person::spouse>);
  attribute set<Car *> * cars
            (inverse<Car::owner>);
  attribute Person *children[];

  instmethod void change_address(
                      in string street,
                      in string town,
                      out string oldstreet,
                      out string oldtown);

  classmethod int getPersonCount();
};

class Car {
  attribute char mark[];
  attribute int num;
  attribute Person *owner
            (inverse<Person::cars>);
};

class Employee extends Person {
  attribute long salary;
  Person *boss;
};
```

This example illustrates all the concepts that we described previously.
The class `Person` is composed of a number of attributes each of one having an interesting particularity.
The `name` attribute is a variable size character array, i.e. a string.
This attribute is *literal*, which means that it has no identifier within a database. The hint `index[]` means that this attribute should be indexed to provide efficient query on the attribute value.

The `age` attribute is a simple *literal* `32-bit integer`.

The `addr` attribute is a *literal* user type attribute. As this attribute is *literal*, the type attribute, `Address`, must have been defined before, which is the case.

The next attribute `spouse` has two interesting particularities:

1. a * character follows the user type *Person*, meaning that this attribute is not a *literal* but an *object* (i.e. with an identifier). The * character means a reference to an object.
2. the hint `(inverse <Person::spouse>)` following `spouse` means that this attribute is a relationship.

As the attribute spouse is not a collection and the target attribute spouse is not a collection, this is a one-to-one relationship.

The cars attribute has also several interesting particularities:

1. as a * character follows the user type, this is an *object* .
2. this attribute is a set whose elements are *object* of type Car . Note that the user type Car is defined afterwards.
3. the hint (inverse <Car::owner >) following cars means that this attribute is a relationship whose target is the owner attribute within the class Car . As the source attribute cars is a collection and the target attribute owner is not a collection, the relationship is a many-to-one relationship.

As indicated by the keyword instmethod , the method change_address is an instance method. Note that this keyword is optionnal as this is the default.
The method getPersonCount is a class method as indicated by the classmethod keyword.

The class Employee inherits from the class Person as indicated by the keyword extends . It introduces two attributes salary , a *literal* integer attribute and boss , an *object* attribute which reference an instance of the class Person . Note that as there is no relationship indication (i.e. inverse keyword), the boss attribute is an object-valued attribute (i.e. a unidirectionnal relationship): in this case, EYEDB does not guarantee the referential integrity.

6. The Object Query Language

EYEDB provides a query language based on the ODMG OQL.
Although EYEDB OQL is not an OML (i.e. an Object Manipulation Language), most of the common language operations can be performed (arithmetic and logical operations, string manipulation, flow control, function definition) as well as query constructs.

EYEDB OQL adds a few features from the ODMG OQL such as flow control (if else , foreach), function definition, an assignement operator, and regular expression operators.

For instance the following example is a EYEDB OQL legal construct:

```
list := foreach x in flatten(1, 2, 10, 24)
            ("alpha_" + string(x));

function max(x, y) (if (x > y) x y);

function fib(n)
   (if (n < 2) n (fib(n-1) + fib(n-2)));
```

Note that the previous code does not perform any query.

The following code perform queries:

```
Person;                          // returns all person
                                 // instances

select x from Person;            // idem

Person.name = "john";            // returns the persons
                                 // whose name is "john"

Person.name ~ "^a.*b";           // returns the instances
                                 // whose name matches
                                 // the regular expression

Person.name !~~ "ja"             // returns the persons
                                 // whose name does
                                 // not matches the regular
                                 // expression in a case
                                 // insensitive way.
Person.age > 2 AND
   Person.age < 10;              // returns persons
                                 // whose age is between
                                 // 2 and 10.

Person.name;                     // returns all person
                                 // names

select x.name from x
       in Person;                // idem

foreach x in (Person)            // for each person
 (if (x.name ~ "^j")             // whose name matches
  x.name := \"_\" +              // the regular expression
       x.name;);                 // "j", adds a "_" before
                                 // the name.

// set the age of the persons whose name
// is "john" to 20:
(Person.name = "john").age := 20;
```

7. The C++ Binding

The C++ binding maps the EYEDB object model into C++ by introducing a generic API and a tool to generate

a specific C++ API from a given schema, built upon the generic API.

Each class in the EYEDB object model is implemented as a C++ class within the C++ API: there is a one-to-one mapping between the object model and the C++ API.

7.1. Transient and Persistent Objects

There are two types of runtime objects: persistent runtime objects and transient runtime objects.
A runtime object is persistent if it is tied to a database object. Otherwise, it is transient.

By default, EYEDB does not provide an automatic synchronisation between persistent runtime objects and database objects.
When setting values on a persistent runtime object, we do not modify the tied database object. One must call the `store` method on the persistent runtime object to update the tied database object.

Note that any persistent runtime object manipulation must be done in the scope of a transaction.

7.2. Example

To illustrate object manipulations, we introduce a simple concrete example using the schema-oriented C++ API, based on the previous ODL example construct:

```
// connecting to the EyeDB server
idbConnection conn;
conn.open();

// opening database dbname
personDataBase db(dbname);
db.open(&conn, idbDataBase::DBRW);

// beginning a transaction
db.transactionBegin();

// creating a Person
Person *p = new Person(&db);

// setting attribute values
p->setCstate(Sir);
p->setName(name);
p->setAge(age);

p->getAddr()->setStreet("voltair e") ;
p->getAddr()->setTown("paris");

// creating two cars
Car *car1 = new Car(&db);
car1->setMark("renault");
car1->setNum(18374);

Car *car2 = new Car(&db);
car2->setMark("ford");
car2->setNum(233491);

// adding the cars to the created person
p->addToCarsColl(car1);
p->addToCarsColl(car2);

// storing all in database
p->store(idbRecMode::FullRecurs);

// committing the transaction
db.transactionCommit();
```

A few remarks about this code:

1. the statement `Person *p = new Person(&db)` creates a transient runtime object. This runtime object is not tied to any database object until the `store` method has been called.

2. all the selector and modifier methods such as `setName` , `getAddr` , `addToCarsColl` have been generated by the EYEDB ODL compiler from the previous ODL construct.

3. the `idbRecMode::FullRecurs` argument to the `store` method allows the user for storing each object related the calling instance: so the runtime object `car1` and `car2` within the `cars` collection will be automatically stored using the `store` method with this argument.

4. the call to `transactionCommit` ensures that the database changes will be kept in the database.

8. The Java Binding

The use of the Java language for an EYEDB binding has been motivated by several reasons:

1. Java is architecture independent,

2. Java is valuable for distributed network environment,

3. Java has a very rich builtin library,

4. Java is secure,

5. Java is easier to program than C++.

The Java binding is very close from the C++ binding: the class interfaces are identical, the functionalities are the same; only the language is slightly different.

The previous C++ code is translated below for the EYEDB Java API:

```
// connecting to the EyeDB server
idbConnection conn = new idbConnection();

// opening database dbname
person.DataBase db = new person.DataBase(dbname);
db.open(conn, idbDataBase.DBRW);

// beginning a transaction
db.transactionBegin();

// creating a Person
Person p = new Person(db);

// setting attribute values
p.setCstate(CivilState.Sir);
p.setName(name);
p.setAge(age);

p.getAddr().setStreet("voltaire" );
p.getAddr().setTown("paris");

// creating two cars
Car car1 = new Car(db);
car1.setMark("renault");
car1.setNum(18374);

Car car2 = new Car(db);
car2.setMark("ford");
car2.setNum(233491);

// adding the cars to the created person
p.addToCarsColl(car1);
p.addToCarsColl(car2);

// storing all in database
p.store(idbRecMode::FullRecurs);

// committing the transaction
db.transactionCommit();
```

As shown in this example, the code is absolutely identical except that that some -> in C++ are replaced by a . character in Java.
The only difference that does not appear in our examples is the object memory management. In the C++ example, one should release all the allocated objects; it is not necessary in Java.

9. The CORBA Binding

The EYEDB CORBA binding is composed of two major components: the first one is a generic CORBA binding; the second one is a schema-driven CORBA binding.

The generic CORBA binding is used to interoperate with any EYEDB database in a generic way. This binding is composed of:

1. a generic and complete IDL interface to the EYEDB System.
2. a generic server which implements this generic interface.

This generic IDL interface is a subset of the C++ and Java generic APIs: it allows the user to perform all the operations that can be performed with the C++ API except a few database administration operations such as copying or moving a database.

The schema-driven CORBA binding is used to work with specific schemas within databases. This binding is composed of:

1. a compiler generating IDL and CORBA implementation stubs from database schemas.
2. a mapping language IMDL (Interface Mapping Definition Language) to perform mapping customizations.

A schema-driven CORBA binding may be used conjointly with the generic CORBA binding.

Because of the full interoperability of CORBA, the generic CORBA binding can be used with any ORB (for instance *Orbix*, *Orbacus*, *VisiBroker*).
The schema-driven CORBA binding generates C++ implementation code for *Orbix* or *Orbacus* ORBs. So to compile the generated code, you need to have *Orbix* or *Orbacus*. Once the implementation code has been compiled and is running, you can use it with any ORB.

In both cases - generic and schema-driven bindings - the architecture is a three-tier architecture based on the following components:

1. the client,
2. the CORBA server: in case of the generic binding, this server does not depend on any schema; in case of the schema-driven binding, the server implementation is generated from the database schema,
3. the EYEDB server.

9.1. The Generic CORBA Binding

The major external component of the generic binding is the EYEDB IDL. The EYEDB IDL is composed of one module (named `EyeDB_ORB`) which includes about forty interfaces and a few literal types (i.e. struct, enum, union and exception).

Each class in the EYEDB object model is bound to an IDL interface. For instance, the `object` class is bound to the `EyeDB_ORB::idbObject` interface.
This interface is as follows:

```
module EyeDB_ORB {
  // ...
  interface idbObject {
    readonly attribute idbClass cls;
    readonly attribute idbOid oid;
    readonly attribute idbDataBase db;

    void store()
      raises (idbException);
    void storeRecMode(in idbRecMode rcm)
      raises (idbException);
    void remove()
      raises (idbException);
  };
```

Using the generic IDL, we introduce a piece of code which creates a person instance, sets his age and name to some given values and store it in the database:

```
// server binding, database opening and
// transaction begin
// [...]

// getting the database schema
EyeDB::idbScheme_var sch = db->sch();

// getting the class Person
// within the schema
EyeDB::idbClass_var person_cls =
  sch->getClassFromName("Person");

// looks for the 'age' attribute
// within the Person class
EyeDB::idbAttribute_var age_field =
   person_cls->getAttributeFromNam e("a ge" );

// looks for the 'name' attribute
// within the Person class
EyeDB::idbAttribute_var name_field =
   person_cls->getAttributeFromNam e("n ame ");

// instantiating a Person
EyeDB::idbObject person = person_cls->newObj();

// set the age to 32
age_field->setInt32Value(person, 0, 32);

// set the name to "john"
name_field->setStringValue(person , "john");

// store the change in the database
person->store();
```

9.2. The Schema-driven CORBA Binding

The previous section shows that the generic IDL interface is not very user friendly to set or get attribute values.

For instance, to get or set the `age` attribute value of an instance of the class `Person`, it would be very nice to dispose of methods such as `getAge` and `setAge`.
With the schema-driven CORBA binding, the user can dispose of such methods and much more.

The input information needed by the schema-driven CORBA binding is a schema: an ODL file or a database containing this schema.
From this information, the compiler generates a schema-driven CORBA bridge composed of:

1. an IDL file,
2. a complete implementation of the generated IDL,
3. a simple sample CORBA server to drive this CORBA bridge.

For instance, from the previous `Person` ODL class, the generated IDL will look like:

```
interface Person {
  attribute string name;
      // mapped from Person::name
  attribute long age;
      // mapped from Person::age
  attribute Address addr;
      // mapped from Person::addr
  // other attributes...
};
```

We say that the generated IDL `Person` interface is `mapped from` the ODL `Person` class.
Each interface attribute is `mapped from` the corresponding ODL attribute.

It is then very simple to manipulate the attributes of an instance of the class `Person`. For example, the previous code becomes:

```
// instantiating a Person
Person_var person = person_factory->
              makePerson();

// set the age to 32
person->age(32);

// set the name to "john"
person->name("john");

// store the change in the database
person->store();
```

9.3. Customizing a schema-driven CORBA Binding

As exposed in the previous section, a schema-driven CORBA bridge can be generated in a quite automatical way from any database schema.
Nevertheless, this last approach is a little bit restrictive in a information system where the external interface that we want to provide is not a one-to-one bridge of the database scheme.
Some attributes stored in the database do not necessary need to be exported in the CORBA interface.
Furthermore, a database schema could be organized in a particular way linked to some implementation constraints for query efficiency.
In a more general case, one should want to provide a CORBA view with a more simple, or more pertinent semantics than the database schema semantics.

EYEDB provides a powerful service to realize a target IDL mapping from one or several ODL schemas.
This service, named IMS (Interface Mapping Service), is based on the IMDL language (Interface Mapping Definition Language) a superset of CORBA IDL.

IMS allows the user to control the mapping for each attribute and method within an IDL interface.

Using the IMS, an IDL interface attribute can be mapped:

- from an ODL attribute,
- from a OQL construct,
- from a C++ expression,
- or not be mapped.

An IDL interface method can be mapped:

- from an ODL method,
- from a OQL construct,
- from a C++ expression
- from arbitraty C++ code,
- or not be mapped.

For instance, let's define an interface MyEmployee mapped from the ODL class Employee in the database and composed of the following attributes:

- his name,
- the name of his spouse,
- his salary converted to euro,
- a boolean indicating if he has children or not,
- a boolean indicating if he his rich,
- his spouse car list.

Here is an IMDL construct implementing the previous specifications:

```
interface MyEmployee from Employee : Person {

  // his name
  map attribute string name
      from name;

  // the spouse name
  map attribute string spouse_name
      from spouse.name;

  // his salary converted to euro
  map attribute long euro_salary
    from expr("salary() / 6.55957") :
    expr("salary(_euro_salary * 6.55957)");

  // an attribute indicating if he has children
  map readonly attribute boolean hasChildren from
    expr("self->getChildrenCount() ? 1 : 0");

  // an attribute indicating if he is rich
  map readonly attribute boolean isRich from
    expr("euro_salary() > 10000");

  // his spouse car list
  map readonly attribute CarList spouse_cars from
    spouse.cars;
};
```

10. Using the EYEDB OODBMS

The EYEDB OODBMS has been used in several projects related to genetics and molecular biology. It was first used at Généthon for the human genome mapping project. It is also in use at Infobiogen and currently undergoes testing in several locations, including industrial companies.

Genomic data represents a good test case for an OODBMS because:

- the data are complex and present numerous cross-references,
- there is a strong hierarchy in the semantic concepts of genomics that inheritance can properly model,
- the genomic databases must manage the huge amounts of data produced by the genome mapping and sequencing projects.
- genomic data are disseminated in a myriad of databases, that should be integrated in some way.

The EYEDB OODBMS is used for several databases:

- the HuGeMap database [7, 9],
- the Virgil database [1, 2, 3],
- the GidB database [6].

HuGeMap contains human genome mapping data, described by a EYEDB schema of 56 classes. It contains more than 1 million of different objects. It is accessible from the EYEDB generic Web server as well as from a CORBA server customized with IMDL. It is interesting to note that the target IDL was defined by the genome mapping community as a standard for data exchange *a posteriori* to the HuGeMap schema [8].

Two graphical interfaces have been developed as clients of the HuGeMap database: MappetShow [13] and ZoomMap [16]. The former was written in Java and makes use of the EYEDB Java API or of the EYEDB CORBA binding. The latter was written with the EYEDB C++ API. These clients are real-time applications that deal with several hundred thousands of database objects.

Virgil contains links that relate objects from two other biological databases (GenBank and the Genome DataBase). Virgil is accessible from the EYEDB generic Web server as well as from a CORBA server.

GidB, the Généthon image database, stores images from the Généthon gene therapy project. Gidb is accessible from a customized Web server.

11. EYEDB and the other OODBMS

Since the beginning of this decade, a lot of OODBMS appeared on the market. Some of them are now well known and are largely used: O2 [4], ObjectStore [10], POET [17], VERSANT [12], ONTOS [5], Objectivity [15]. We try to give here a few comparison elements between EYEDB and these OODBMS.

11.1. Functionalities Comparison

EYEDB provides currently less functionnalities than some of these OODBMS. For instance, EYEDB does not provide versioning and replication functionalities and no Graphical User Interface are offered except a primitive WEB based tool allowing the browsing of any database.
On the other hand, EYEDB put priority on languages: it provides an extended ODMG OQL, an extended ODMG ODL, a powerful interface mapping language IMDL to define and generate CORBA views, a C++ and a Java binding. Furthermore, EYEDB OQL is currently being extended to become an actual programming language (i.e. both a manipulation and query language).

11.2. Performance Comparison

As no standardized benchmarks have yet been run on EYEDB, it is somewhat risky to say that EYEDB is faster than so-and-so other OODBMS. However, EYEDB has been designed to be as efficient as possible: efficiency and lightness are its leading features:

- EYEDB is built on a every efficient and lightweight memory mapped storage manager,
- this storage manager is currently being adapted to take fully benefit from the 64-bit architecture,
- it provides a local mode access which allows the client to map database objects directly in his virtual space avoiding so any buffer copies at reading,
- it provides also a transaction less mode to access object in a read only databases avoiding the overhead of object locking and transaction computing.

Benchmarks will be run for comparison with other OODBMS.

12. Conclusion

EYEDB is not a major breakthrough in the conception of database systems but is rather a synthesis of state of art techniques driven by application requirements and taking advantage of the present and forthcoming evolutions of hardware and operating system. For example, it will greatly benefit from the 64-bit architecture since it is based on a virtual memory mapping strategy.

EYEDB allows for the management of large databases with a complex data schema and offers very good performance to access data.
EYEDB follows the ODMG specifications and can distribute objects using the OMG CORBA standard.

EYEDB is now about 200,000 lines of C++ code and Java code. It runs on Solaris 2.x Sparc platforms and will be ported to Linux during the first semester of 1999.

You can get more information on the OODBMS EYEDB at the EYEDB home page [19]. This page contains the full online programming manual, links to related publications and a trial version for Solaris can be downloaded.

13. Acknowledgements

The EYEDB OODBMS [19] has been developed at Sysra Informatique since 1994 in collaboration with the GIS INFOBIOGEN, with funding from the ANVAR, the Conseil Régional d'Ile de France and from the European Commission *(BIO4-CT96-0346)*.

References

[1] F. Achard and E. Barillot. Virgil: a database of rich links between GDB and GenBank. *Nucleic Acids Research*, 26(1):100–101, 1998.

[2] F. Achard, C. Cussat-Blanc, E. Viara, and E. Barillot. The new Virgil database: a service of rich links. *BIOINFORMATICS*, 14(4):342–348, 1998.

[3] F. Achard, G. Vaysseix, P. Dessen, and E. Barillot. Virgil database for rich links (1999 update). *Nucleic Acids Research*, 27:113–114, 1999.

[4] M. Adiba and C. Collet. *Objets et bases de données, le SGBD O2*. Hermès, 1993.

[5] T. Andrews and all. *The ONTOS Object Database*. Ontologic, Inc, Burlington, Massachusetts, 1989.

[6] N. Armande. Gidb, the Généthon image database. http://www.genethon.fr/bdimage/bdimage.html.

[7] E. Barillot, F. Guyon, C. Cussat-Blanc, E. Viara, and G. Vaysseix. HuGeMap: a distributed and integrated Human Genome Map database. *Nucleic Acids Research*, 26:106–107, 1998.

[8] E. Barillot, U. Leser, P. Lijnzaad, C. Cussat-Blanc, K. Jungfer, F. Guyon, C. H. G. Vaysseix, and P. Rodriguez-Tomé. A proposal for a CORBA interface for genome maps. *BIOINFORMATICS*, 15, 1999.

[9] E. Barillot, S. Pook, F. Guyon, C. Cussat-Blanc, E. Viara, and G. Vaysseix. The HuGeMap database: Interconnection and Visualisation of Human Genome Maps. *Nucleic Acids Research*, 27:119–122, 1999.

[10] C. Lamb et al. The objectstore database system. *Communications of the ACM, 34(10)*, pages 50–63, 1991.

[11] C. G. G. Cattell and al. *Object Database Standard, ODMG 2.0*. Morgan Kaufmann, 1997.

[12] V. Corporation. Versant Corporation. http://www.versant.com/.

[13] F. Guyon. Mappetshow, a viewer for very dense maps. http://www.infobiogen.fr/services/Mappet/MappetShow.html.

[14] H. F. Korth and A. Silberschatz. *Database system concepts*. MacGraw-Hill, 1991.

[15] Objectivity. Welcome to objectivity. http://www.objectivity.com/.

[16] S. Pook, G. Vaysseix, and E. Barillot. Zomit: biological data visualisation and browsing. *BIOINFORMATICS*, 14:807–814, 1998.

[17] P. Software. Data management for the Internet Age. http://www.poet.com/.

[18] M. Stonebraker and J. M. Hellerstein, editors. *readings in database systems*. Morgan Kaufmann, 1998.

[19] E. Viara. The EYEDB Home Page. http://www.sysra.com/eyedb/.