The O_2 Database Programming Language

C. Lécluse and P. Richard

GIP Altaïr - Domaine de Voluceau, B. P. 105, 78153, Rocquencourt France

Abstract

In this paper, we describe the O_2 database programming language as it is currently implemented. We first show how O_2 provides the user with both objects and complex values. Then, we present the persistence management of O_2. We describe how objects are encapsulated and manipulated through methods and how values are directly accessible through operators. We also present the subtyping and inheritance relationships in O_2 together with the type-checking mechanism. Finally, we mention some interesting features which deal with exceptions and we make a comparison between O_2 and several other object-oriented database systems.

1 Introduction

The major objective of Altaïr is to prototype a complete development environment for data intensive applications. The functionalities of such a system should include those of a DBMS, those of a programming language and those of a programming environment. We decided to build an object-oriented database system, named O_2, and its programming environment. Our motivations for this choice are the following:

- We do believe that one of the main bottlenecks to the productivity of the application programmer is the impedance mismatch between the programming language and the database. This impedance mismatch cannot be solved by redefining the database box (i.e. by changing the frontier between the programming language and the database system) but by mixing database technology and programming language technology to build a complete system with the functionalities of a DBMS and of a programming language.
- We do believe that, among the available technologies produced by programming language people and among the possible approaches, the object-oriented approach is the best one to mix with database technology. This is due both to the intrinsic characteristics of the approach and to the appeal this paradigm has to programmers.

The next choice concerned the programming language of the system. Among the possible solutions (extending an existing language, designing a new language, or being language independent), we have chosen the last one mainly for marketing reasons (from a purely technical point of view the second was probably the best). The system is viewed by the user as consisting of a data definition language (DDL) by which the user can manipulate a hierarchy of classes. He/she can attach methods to classes or to objects by writing these methods in various languages. Our first target set of languages consists of C and Basic. Rather than speaking of the O_2 database programming language, one can think of the O_2 database programming languages. Programming in O_2 is done in two distinct steps. First, the programmer defines classes using O_2 commands. Then, he/she programs the code of his/her methods using one of the O_2 programming dialects. For the time being, two programming languages are specified, the CO_2 language which relies on C and the $BasicO_2$ language which relies on Basic. In this paper, we specifically report on the merge of programming language technology and database technology [AtB 87] and we shall only describe CO_2, as it is the first we have implemented. Furthermore, the approach followed for $BasicO_2$ is similar to that of CO_2. A full description of the O_2 object manager can be found in [VBD 89].

This paper is organized as follows. Section 2 contains an informal presentation of objects and values in O_2. Section 3 describes the data organization in O_2 through examples. Section 4 shows how programming is done and illustrates it using CO_2. Section 5 explains how inheritance works in O_2 and justifies its foundations through subtyping. Section 6 illustrates the features of O_2 which deal with sets and exceptions. Section 7 briefly describes how methods are type-checked in O_2 and how method safety is insured. Section 8 compares the O_2 system with several other object-oriented database systems (OODBS). Finally, we present some conclusions.

2 Objects and Values in O_2: an Informal Presentation

O_2 is object-oriented: this means that information is organized as *objects* which have an identity and encapsulate data and behaviour. Manipulation of ob-

jects is done through *methods*, which are procedures attached to the objects. Object identity is useful for supporting object sharing and updates management. The theoretical foundations for object identity as a programming language primitive can be found in [AK 89]. In classical object-oriented languages such as Smalltalk [GR 83], the value encapsulated in an object is always an atom or a tuple of other objects. In object-oriented database systems, this value is classically a tuple or a set of objects [MOP 85], [Ban et al 87], [Kup 85], [LR 88] since databases must provide flexible management of large sets of data. However, this value is always a flat value, as it can only contain identifiers of other objects, and not directly other complex values. This limitation is exactly the same as the limitation of relational systems which has motivated the introduction of nested relations and complex objects. In O_2, we provide the user with the possibility of defining, not only objects, but also *values* [1] as in standard programming languages or in the so-called complex objects[2] languages [AB 87], [Kup 85], [BK 86]. Of course, complex (nested) structures can always be modeled through the use of identifiers but we think that this solution is awkward, just as the modeling of nested relations with surrogates in relational systems. For example, we can consider the following three objects (objects identifiers are written in italics):

eiffel_tower:
tuple(**name**: "Eiffel tower",
address: *eiffel_address*,
description: "Paris famous monument",
admission_fee: 25 FF)

eiffel_address:
tuple(**city**: *paris*,
street: "Champs de Mars")

paris:
tuple(**name**: "Paris",
country: "France",
population: 2.6)

While both *paris* and *eiffel_address* where modeled as objects in this example, we believe that they should be treated differently: *eiffel_address* is nothing more than a pair of strings which only appears in the value of *eiffel_tower*. On the other hand, cities evolve with time (think of the population) and might be shared by other monuments, therefore we wish to model *paris* as an object. In our system, the object *eiffel_tower* should be modeled as follows, with address appearing as a *structured value* and *paris* as an *object*:

eiffel_tower:
tuple(**name**: "Eiffel tower",
address: **tuple**(**city**: *paris*,
street: "Champs de Mars"),
description: "Paris famous monument",
admission_fee: 25)

An object-oriented data base system intends to provide the application programmer with a powerful applications development support using encapsulation and inheritance. This important step forward should not be canceled by an increased complexity in structure manipulations and by losing data independence due to the navigation through objects identifiers. We claim that "pure" object-oriented database systems have severe drawbacks. The user has to define a new class every time he/she needs a complex value. This results in an undesirable growth of the class hierarchy.

Thus, the class hierarchy should only contain classes which correspond to data shared by distinct software modules. It should not be polluted by classes which are only used to describe non shared values.

Some authors already felt the need of dealing with both objects and values: in [Ban et al 87], there is a notion of exclusive relationship between an object and some of its components. When an object is exclusively owned, it can not be shared. The same notion is introduced in [CDV 88] where the programmer can specify whether he deals with a reference to a complex value or with the complex value itself, and also if an object can be shared by several objects or is exclusively owned by an object. This proposition introduces the desired distinction, but values are still implemented and manipulated as objects. In the new version of FAD [DKV 87], one can manipulate objects and values. A value in FAD is either atomic or structured. A structured value contains values. An object in FAD has an identifier and a state. A state is either an atomic value or a structure containing objects and values. Objects may be updated while values may not. Objects also allow for sharing. Exodus [CDV 88] also provides the user with a mix of objects and values similar to us (see Section 8). In O_2, we follow a similar approach. The user may choose between two kinds of organizations: *classes* whose in-

[1] The previous prototype of O_2 only dealt with objects [LRV 88]

[2] which are not objects in the object-oriented terminology but rather complex values

stances are objects and which encapsulate data and behavior and *types* whose instances are values. Values are not encapsulated, that is, their structure is known by users and they are manipulated by operators. To every class is associated a type, describing the structure of its instances. Classes are created explicitly using commands and are parts of the inheritance hierarchy. Types are not created explicitly since they only appear as components of classes and do not appear in the inheritance hierarchy. The underlying model is presented in [LR 89] and analyzed in [AK 89].

3 Types and Classes

In O_2, the user has two ways of structuring data: types and classes. Types are recursively constructed using atomic types such as integers, floats, strings, class names and the set, list and tuple constructors. Instances of types are *values*. These types are similar to classical types in programming languages. The following expression is an O_2 type:

tuple (**name**: string,
country: string,
population: float,
monuments: **set**(Monument))

This type describes cities. The **monuments** attribute has a set structured value. Monument is a class name. A *value* of this type can be:

tuple(**name**: "Paris",
country: "France",
population: 2.6,
monuments: **set**(*eiffel_tower, triumph_arch*))

Recall that we use italics to denote objects. One can see from the above example that values can be arbitrarily complex. The elements of the set value of the attribute "monuments" are objects as we shall show in the next subsection. In O_2, the user builds types using atomic types such as string, float, integer, char and boolean and three type constructors: tuple, set list. There is no restriction on the use of these constructors. We have already used the set and tuple constructors, an example of use of the list constructor is given in the next subsection.

3.1 The Schema Definition Language

In O_2, the schema is a set of *classes* related by inheritance links (see Section 5) and/or composition links. A class describes the structure *and* the behaviour of a set of objects. The structural part of a class is a type as defined above and the behavioural part is a set of methods (see Section 4). Classes are created using schema definition commands as follows:

add class City
type tuple(**name**: string,
country: string,
population: integer,
monuments: **set**(Monument))

add class Monument
type tuple(**name**: string,
address: **tuple**(**street**: string,
city: City),
description: string,
closing_day: **list**(string),
admission_fee: integer)

We denote class by capitalizing the first letter. The first class has a name "City" and a type which is given after the keyword **type**. Instances of this class are objects. That is, they have a unique internal identifier and a value which is an instance of the type associated to the class. Objects are encapsulated, that is, their value is not directly accessible and they are manipulated by methods as explained in Section 4. The second class defines historical monuments. Note that classes can be mutually referencing. The "City" class references the "Monument" class which in turn references the "City" class. For every "Monument" object, the value of the "**city**" attribute is an object which may itself references the "Monument" object.
Following an approach similar to Galileo [ACO 85], the equivalence of classes is by name as opposed to type equivalence which is by structure. That is, the type of the values only depends of their structure. One the other hand, two classes are always distinct and the compatibility rule is the name equivalence rule.

3.2 Object Creation

Creation of objects is done through a system command called "new". This command "new" takes as input the name of the class corresponding to the object to be created. The object is created with a default value depending on the type associated to the class. The default values are: the empty string, the integer 0, the float 0.0, the empty list and the empty set for list and set types, and a tuple of default values for tuple types.

3.3 Naming and Persistence

In O_2, objects or values can be named. The following is an example of naming:

add object Eiffel_tower: Monument

The name Eiffel_tower will then stand for an object of class Monument. In the same way, one can name a value as follows:

add value Paris_monuments: set(Monument)

"Paris_monuments" is a name for a value of type (set(Monument)). In O_2, persistence is attached to names, that is, every named object or value is persistent. Such a name can be seen as a global variable dynamically attached to a given object or value and makes it persistent. The attached object can be changed by assignment. For instance, we can write:

Eiffel_tower = **new**(Monument)

This instruction assigns a newly created object to the name "Eiffel_tower". The initial value of the object is the tuple default value corresponding to the type. This object will always be accessible trough the name "Eiffel_tower" during the life of the system, except if the user makes another assignment.

The persistence rules are the following:

1. every named object or value is persistent,
2. every object or value which is a part of another persistent object or value is persistent.

For example, let us assume that we have made the following assignment:

Paris_monuments =
set(Eiffel_tower, *triumph_arch*)

where Eiffel_tower is a named object and *triumph_arch* denotes an object of class Monument with no name. Then, these objects are persistent. The first one is already persistent due to its name and the second is persistent because it is an element of the named value. The same holds for objects or values which appear as an attribute value in the named object "Eiffel_tower".

The *extension* of a class is the set of all objects created using the **new** command applied to that class. The system provides the user with an automatic management of class extensions. This is done using a set value which collects all the objects of a class. For instance, one can write:

add City **with extension**
type tuple(**name**: string,
country: string,
population: integer,
monuments: **set**(Monument))

The **with extension** clause in the class definition tells the O_2 system to create a named value of type "set(City)" with name "City". Moreover, every city created with the "**new**(City)" command will be automatically inserted in this set and will thus persist, as it is a component of a persistent set. Note that, according to our persistence rules, objects of a class without extension will not persist unless they are explicitly named or components of some other persistent object or value. Classes with no extension are a natural way of dealing with transient objects.

4 Objects and Values Manipulation

4.1 Methods Definition

In the object-oriented approach, objects are manipulated by *methods*. A method is a piece of code which is attached to a specific class and which can be applied to objects of this class. In O_2, method definition is done in two steps. First, the user declares the method by giving its signature, that is, its name, the type or class of the arguments and the type or class of the result if there is one. Then, he/she gives the code of the method. The following is a method declaration:

add method increase_fee (amount: integer)
in class Monument

This method increases the **admission_fee** field of a Monument object. Methods can be *private* or *public*. Private methods are only visible within their class, i.e. in the methods attached to that class. Public methods are visible by every classes and can be freely used. When declaring a method, the user can add the keywords **is public** in order to make it public. The default is private.

O_2 follows a multi-language approach. This means that method programming is done in a standard programming language such as C or Basic with manipulation of O_2 objects and values. The main idea is that most of the programming is done using the programmer's favorite language. This includes iterations, control structures and arithmetics. Access to, and manipulation of, objects and values is done using O_2 features. We give below, as an example,

the code of the method "increase_fee" using the CO_2 language.

```
body increase_fee(amount: integer) in class Monument
        co2 { (*self).admission_fee += amount; }
```

The curly brackets delimit the CO_2 block of code as in pure C. The value of an object is obtained using the "dereferencing" method *, thus "self" is the object and "*self" is the associated value. This method is applied using a special syntax which follows the C "*" use. It illustrates the association between objects and values. As in standard programming languages, objects can be seen as pointers to values. In the example, the value "*self" is tuple-structured, and the access to an attribute is done using the dot operator. The assignment is done as in C and increments the integer value representing the admission fee. Notice that we stick to the C syntax for manipulating O_2 values such as dereferencing or extracting a tuple field. This way of manipulating objects is syntactically very close to what is done in C++ [Str 86]. In O_2, however this similarity is purely syntactical, as objects and values are implemented and manipulated in a special way by a persistent object manager and the CO_2 compiler generates calls to this object manager [VBD 89].

A method is applied to an object by *message passing* whose syntax is the following:

[receiver selector(arguments)]

The square brackets are used to delimit O_2 message passings. "receiver" denotes an object to which the method whose name is "selector" is applied. This eventually returns an object depending on the method code. For example, "increase_fee" is applied to a monument using the message passing:

"[Eiffel_tower increase_fee(3)]".

The keyword "self" in the above code will denote the object "Eiffel_tower" when the method is applied.

4.2 Manipulating Values

The CO_2 language allows the construction of O_2 values using the set, list and tuple constructors. We can, for instance, write a set value containing four integers as follows: set(1, 4, 34, -21). The following associates a value to a newly created object:

```
Eiffel_tower = new(Monument);
*Eiffel_tower =
tuple(name: "Eiffel tower",
      address: tuple(city: paris,
                     street: "Champs de Mars"),
      description: "Paris famous monument",
      closing_day: list("Christmas", "Easter");
      admission_fee: 25)
```

This assumes the * method is public for the Monument class. We have seen above that we can extract a field of a tuple value using the dot operator. All the CO_2 value manipulations are done in this way, using the classical C constructs. For instance, we shall append elements to the **closing_day** list of the Eiffel tower or modify one entry of the list as follows:

```
*Eiffel_tower.closing_day += list("June 6th");
*Eiffel_tower.closing_day[1] = "January 1st";
```

O_2 provides the user with the usual sets and lists operators (union, intersection, difference, cardinality, concatenation, ...) whose syntax follows as much as possible the C syntax.

4.3 Iterator

The iterator described here is applied on set or list structured *values*, not on objects. Indeed, objects are encapsulated and one should not know what is the structure of the encapsulated value. Of course, the values to which the iterator is applied may be a set (a list) of values or a set (a list) of objects. CO_2 provides the user with an iterator which allows for easy sets or lists manipulations.

for (x **in** S [**when** condition]) <Statement>.

This is an extension of the classical C iterator. It applies the given statement with the variable x bound to every element of the set (or list) value S satisfying the optional condition. The **when** clause adds no power to the **for** iterator, but allows some optimization when the condition is directly evaluable by the object manager. For instance, we can write:

```
co2{ o2 Monument x;
     for (x in Paris_monument
     when (*x.admission_fee ≤ 20.00))
     [x increase_fee(amount)];
     }
```

The above code increases the admission fee of all the monuments located in Paris, whose **admission_fee** is less or equal to 20.00FF. The expression "**o2** Monument x" declares an O_2 variable which is used to denote objects of class "Monument". Recall that "Paris_monument" is a named value of type **set**(Monument) which is supposed to contain all monuments of Paris. The **for** iterator is of course less concise but more

flexible than the classical join operation. The reader should notice that it is far more powerful in the context of O_2 which is a programming language and not an end-user query language.

5 Subtyping and Inheritance

Inheritance is a powerful mechanism which allows the user to define classes in an incremental way by refining already existing ones. O_2 provides the user with an inheritance mechanism based on subtyping.

5.1 Subtyping

Subtyping is a semantic relationship which connects two types. There are several ways of defining subtyping. In O_2, we defined a set inclusion semantics for subtyping. That is, a type is a subtype of another if and only if every instance of this type is also an instance of its supertype. This allows to say that a person is a human or that an employee is a person. The formal definition of the O_2 type system is given in [LR 89]. Another approach is taken by Vision [Car 87]. In this system, subtyping is expressed by means of a mapping from the objects of the subtype to objects of the supertype. We adopted a Cardelli-like approach [Car 84] for tuple subtyping. A tuple type is a subtype of another if it is more defined, that is, if it contains every attribute of its supertype plus some new ones and/or refines the type of some attributes of its supertype. The following example illustrates this.

tuple(**name**: string,
  address: **tuple**(**street**: string,
    city: City),
  description: Text,
  closing_day: **list**(string),
  admission_fee: integer,
  number_rooms: integer,
  rate: integer)

is a subtype of:

tuple(**name**: string,
  address: **tuple**(**street**: string,
    city: City),
  description: Text,
  closing_day: **list**(string),
  admission_fee: integer)

Another characteristic of this subtyping relationship is that a set-structured type "set(T)" is a subtype of "set(T')" if and only if T is a subtype of T'. For instance:

set(**tuple**(**name**: string,
  address: string))

is a subtype of:

set(**tuple**(**name**: string))

The same relationship holds for lists.

5.2 Inheritance

Based on this subtyping relationship, O_2 offers an inheritance mechanism. We can define the Historical_hotel class as follows:

add class Historical_hotel **inherits** Monument
  type tuple (**number_rooms**: integer,
    rate: integer)

The effect of this declaration is the definition of an Historical_hotel class whose associated type is a subtype of the Monument type. The user only has to give the extra attributes (the other ones are taken from the definition of inherited class). The O_2 command interpreter checks whether the inheritance definition is legal, that is if there is no subtyping violation, and creates the subclass according to the subtyping rules. An object of class Historical_hotel will automatically be considered as an object of class Monument. This results in the possibility of applying any method of class Monument to Historical_hotel objects. O_2 also allows for multiple inheritance, as shown below. We first define a "Restaurant" class.

add class Restaurant **with extension**
  type tuple (**name**: string,
    address: **tuple**(**city**: City,
      street: string),
    menus: **set**(**tuple**(**name**: string,
      rate: float)))
add method check_rates(float): boolean
  in class Restaurant

The method "check_rates" checks whether the menus rates are less than a given amount. We can now define an "Historical_restaurant" class as follows:

add class Historical_restaurant **with extension**
  inherits Monument, Restaurant
  type tuple (**redefines name**: string,
    redefines address:
      tuple(**city**:City,
        street: string))

add method check_rates(float): boolean
  in class Historical_restaurant

Class "Historical_restaurant" inherits both from Monument and from Restaurant. Here, the method "check_rates" checks whether the menus rates are less than twice the amount[3]. We shall not detail the conditions that method signatures must satisfy in order to be inherited through the subclasses, see [LRV 88].

As opposed to single inheritance, possible ambiguities may arise with multiple inheritance when an attribute or a method name is defined in two or more superclasses. There are several solutions to such ambiguities [Ban et al 87], [SCBKW 86]. We decided to follow an approach similar to that of Trellis/Owl [SCBKW 86]. That is, the user has to explicitly redefine the attribute or method name when needed. We think that, as opposed to solutions where the system solves the ambiguity by itself by ordering the superclasses, this solution is more natural and enhances the readability and maintainability of the schema. Thus, the "Historical_restaurant" class redefines the attributes "name" and "address" which are both present in the classes "Restaurant" and "Monument". The reader should note that we do not infer the subclass relationship which is user defined. The system just checks whether it is legal with respect to the subtyping rules.

5.3 Late Binding

An important feature of object-oriented systems, which is fully implemented in O_2, is late binding. The actual code of a method to be executed is not selected at compile-time but at run-time depending on the actual type of the receiver object. The main benefit is dynamicity and reuse of existing software. Indeed, existing methods do not have to be recompiled when the code of the methods they use is changed. An example of use of late binding is:

```
for(x in Restaurant) {
if(![x check_rate(120.50)])
  printf("restaurant %s is expensive", *x.name);
}
```

This iteration loop applies the "check_rate" method to every restaurant. Due to our subtyping semantics, some of them are historical restaurants. For these ones, the system automatically applies the method defined in the class "Historical_restaurant". This avoids to explicitly take into account the different status of historical restaurants. Late binding is a critical operation from the performance point of view. The O_2 choice for the implementation of late binding is described in [VBD 89].

6 Interesting Features

In this section, we describe some interesting features of O_2 which improve the expressibility of the O_2 language.

6.1 Exceptional Attributes

Due to the semantics of the subtyping relationship, a tuple value can have extra attributes. If we consider the Monument class, the "Eiffel_tower" object can have a value which also contains an attribute "height". This extra attribute will not be dealt with by the methods associated to the Monument class, however, the standard operators available on tuple values will handle it. For instance, the following is a correct CO_2 code:

```
{Eiffel_tower = new(Monument);
 *Eiffel_tower=
 tuple(name: "tour Eiffel",
       address: tuple(street: "Champs de Mars",
                      city: Paris),
       description: "Paris famous monument",
       closing_day: list("Christmas", "Easter"),
       height: 315,
       power: 15.5);
 (*Eiffel_tower). height = 320;
 return ((*Eiffel_tower).height);}
```

Assuming that de-encapsulation is allowed on Monument, this code first modifies the value of "Eiffel_tower" and adds a "height" and a "power"[4] attribute. Then, using the dot operator, the height attribute is updated and finally its new value is returned. Note that exceptional attributes are allowed for any tuple object or value, even if not named.

6.2 Exceptional Methods

One can associate specific methods to named objects. These methods are used to characterize the exceptional behaviour of an object. One can also override an existing method in the class of the object with an exceptional method. An example of this mechanism is given below:

[3] The restaurant is historical and is allowed to increase its rates!

[4] The Eiffel tower is also a radio and a TV broadcasting station

add method increase_power (amount: float)
in object Eiffel_tower

This method will be used to increase the broadcasting power of the Eiffel tower station. Note that the method is associated to the name not to a particular object, and that the actual object associated to the name "Eiffel_tower" can change at run-time. The late binding process will associate the exceptional method to the object currently bound to the name.

7 Type-Checking

O_2 emphasizes user defined classes and their associated types. This is a natural way to structure data. An other important motivation is type-checking. The goal of O_2 is to increase the productivity of business applications programmers. In this context, safety of programs is critical. Thus, O_2 offers a static type-checker which detects the illegal manipulations of O_2 objects and values when inheritance is not used. When full use of inheritance or of exceptional attributes is done, O_2 must perform some run-time type-checking. Of course, since the method code of O_2 can be written in several languages which may be loosely typed such as C, there also may be errors due to the host languages manipulations. The type checking algorithm used in O_2 is standard. It is conceptually similar to that of Trellis/Owl [SCBKW 86] in that a variable can only be assigned values (resp objects) of its declared type (resp class) or of any subtype (resp subclass). The user may modify the schema dynamically. In this case, a method which has not been recompiled may perform message passings which reference non existing methods. Of course, if the user recompiles every method which may be concerned by the schema modifications, references to non existing classes or methods are detected by the type-checker. Other run-time errors occur with exceptional attributes. At compile-time, the type-checker may not know whether an attribute, which is not present in the variable declaration, but is referred to in the code, is an exceptional attribute or not. Accepting such a manipulation implies that the method may fail at run-time because the actual value does not possess this exceptional attribute. We accept this for the sake of expressive power. The user may choose not to use exceptional attributes and have safe programs.

8 Related Works

In this section, we list the main characteristics of O_2 and see what kind of solutions others OODBS's provide. We compare O_2 to other systems on the basis of the programming language only. We shall not be concerned by query facilities, user interface or physical management.

Gemstone [MOP 85] is to our knowledge the first implementation of an OODBS. The philosophy of Gemstone was to turn Smalltalk into an database system without significant modifications of the Smalltalk programming language. Vision [Car 87] is another interesting approach. Vision models data in a way similar to Daplex [Sch 81]. All informations about an object are embodied in *functions* which map a collection of objects into another. However, function application follows a message passing mechanism using a Smalltalk-like approach. Iris [DFKLR 86] also follows a functional approach in that, to every object, is associated a set of functions which characterize its content. Orion [Ban et al 87] is another example of a functional approach since it is implemented using Lisp and has a Lisp syntax for the message passing. Vbase [AH 87] follows an approach similar to O_2 as the corresponding language (COP) is a strict superset of the C programming language. Although it is not a true OODBS, Trellis/Owl is another example of an object-oriented language with an imperative way of programming. It has a conventional programming language syntax and uses a procedure call notation to invoke operations on objects. Trellis/Owl does not have all the database functionalities but provides persistence through an object repository. A common characteristic of these approaches is that they provide compile-time type-checking. The Exodus system [CDV 88] is also an object-oriented system which allows abstract data types definitions, objects and values and a query language named Excess. Programming is also done in the E language which is a persistent C++.

We now list the main original features of O_2 and describes what is done in other OODBS's.

- O_2 ***provides the user with both objects and structured values.***
  We do not follow a pure object approach as in Smalltalk or Gemstone but allow the definition of nested values built using the set, list and tuple constructors. O_2 manipulates objects using methods and values using operators. That is, full object-oriented features are available for objects (such as late binding and inheritance of methods) and values are manip-

ulated as in database systems. Most OODBS's provide object constructors similar to the set and tuple constructors. The Exodus system also gives an array constructor which is similar to the O_2 list constructor. The distinction between objects and values can also be found in Orion. In this system, however, the notion of complex value is implemented as a *dependent object* [KBCGW 87]. That is, non shared values are still objects with a constraint enforcing their privacy. The Exodus data model [CDV 88] also provides the user with this distinction. However, just as in Orion, values are second-class objects with no identity. In O_2, we enforce the distinction between objects and values in the programming language because we encapsulate objects which can only be manipulated through methods. Exodus adopts a point of view which is less object-oriented but more database oriented. In Exodus as in Orion, for the sake of query simplicity and uniformity, objects and values are manipulated in the same way. In Iris and in Vision, one only has objects. Objects are atomic items which can be printable (like the object "3") or not. If an object is not printable, its value is characterized by a set of functions which can be stored, and thus plays the role of attributes, or computed. As opposed to O_2, where the three object constructors have exactly the same rights, due to their approach, Iris and Vision manipulate complex objects which are records of functions which can however be multi-valued.

- O_2 *follows a multi-language approach.*
  Classes and types are created using the O_2 schema commands, but the code of methods can be implemented using several O_2 extensions. In this paper, we concentrated on CO_2 but another extension is currently under implementation based on the Basic language. Up to now, among the existing OODBS's, O_2 is the only multi-language system.

- O_2 *has a compile-time type-checker.*
  Systems such as Gemstone based on a Smalltalk like approach do not provide such a functionality nor do systems based on Lisp such as Orion. On the other hand, systems based on an imperative paradigm are statically typed. Among them, let us quote Trellis/Owl and Vbase. As in O_2, Trellis/Owl and Vbase have a strong typing. That is, every object is an instance of a type and every variable is declared of a type. A variable can only be assigned objects of its type or of a subtype of its type. In order to have statically typed languages, types and methods are not modeled as objects and manipulated by methods but are primitive constructions manipulated by schema commands.

- O_2 *provides an automatic management of persistence through named objects and values.*
  Every named object or value is persistent and every component of a persistent object or value is itself persistent. The name can be seen as a handle which allows the user to access an object or value after the end of a program which has defined it. Other systems provide a somehow similar way of managing persistence. Objects in Orion also persist because they are components of persistent collections. For every user defined type, the system generates a set structured class which has at least one instance which groups the instances of the former class. In Gemstone, the management of persistence also uses reachability informations, that is, objects are persistent if they are attached to a persistence root or another persistent object. An Exodus database is a collection of named persistent objects.

- *Updates are always implicit in* O_2.
  Objects are created using the "new" command. If a class is created "**with extension**" then a named set value is created which will contain every object of the class which will thus persist. If the class is not created with extension, then the created objects will only persist if attached to other persistent objects. Deletion of objects or values is obtained by removing the links which attach them to the persistence roots (the names). Classes with extension are also provided with a "delete" method which allows objects to be removed from the class extension when no other objects or values refer to it. Gemstone and Orion have a similar update policy, as they have a similar persistence policy. In Vbase, however, every object is persistent, and temporary objects have to be deleted explicitly.

- O_2 *has a set inclusion semantics for subtyping.*
  Objects of a subclass are objects of the superclasses. For instance, if one performs a display on the instances of Monuments, one will also see the instances of Historical_hotel. Some systems follow this approach, such as Trellis/Owl

and Iris. On the other hand, Vision has a mapping semantics: an object of a subclass has a corresponding image object in its superclass. We find this somehow unnatural. However, this provides the same kind of functionalities at least in the context of single inheritance as provided in Vision. Iris has also a set inclusion semantics. This is even more general, since an object can have several types even if these types are not related in the specialization hierarchy. We are not aware of the way they solve ambiguities. Orion has no set inclusion semantics.

- *Multiple inheritance conflicts are solved by users.*
  Trellis/Owl proposes a similar solution. The user must solve the ambiguities which may arise. For instance, when there is an ambiguity on the inheritance of a method (operation in Trellis/Owl), the user must specify which one he/she wishes to inherit or redefine it. O_2 follows exactly the same approach as shown in Section 5. Another system which provides multiple inheritance is Orion. As opposed to O_2 or Trellis/Owl, Orion automatically solves ambiguities. Roughly speaking, the system maintains an ordering among the superclasses which desambiguates inheritance of methods. To our knowledge, other systems, such as Gemstone or Vbase do not support multiple inheritance.

- *The O_2 system allows exceptional methods and attributes for objects.*
  Exceptional methods can be associated to names. These methods are only accessible from the object currently attached to the name and override the methods of the class. Exceptional attributes can be added to every tuple structured objects or values. To our knowledge, no other OODBS provides such a functionality.

Another interesting approach is that of Galileo [ACO 85]. We did not put it in the collection of items above since it is not really an object-oriented data base management system, however it has some object-oriented features such as classification, abstract types and types hierarchies. As opposed to O_2, Galileo does not have the set constructor but is higher order and has a function type constructor. It has the notions of concrete and abstract types which roughly correspond to our types and classes. Galileo presents a very interesting solution to persistence which is however not yet implemented to our knowledge. Another important difference with O_2 is that Galileo does not support object identity. We did not put in this list the Damokles database system [DGL 87] which is designed for software engineering environment. As their designers say, Damokles is a "structurally object-oriented" database. That is, Damokles provides the user with object identity, complex objects based on the tuple constructor and n-ary bidirectional relationships between objects. However, Damokles does not provide encapsulation, inheritance or late binding. It is rather an " complex objects" system.

9 Conclusion

In this paper, we have described the features of the O_2 system as it is currently running. We only described the CO_2 programming language, but most of the described features are common to both CO_2 and $BasicO_2$, and the difference between the two languages is mainly syntactical.

The target applications for our language are (i) traditional applications such as business and transactional (excluding however very high performance transaction processing systems), (ii) office automation applications and (iii) spatial data management (such as geographic data management). At this stage of the game, no specific emphasis is given to CAD/CAM, CASE or knowledge base applications, but we believe that, in a later stage, the system could be enhanced to serve also these applications.

Altaïr started in September of 86. We first implemented, in December 87, a throw away prototype [Ban et al 88] whose data model is described in [LRV 88], in order to test and show the functionalities of the system.

This gave us a lot of feed back and we completely redesigned the system, its language, its data model [LR 89] and its architecture. The major differences between this version and the throw away prototype from the language point of view are: (1) complex values together with objects, (2) names for objects and values, (3) the list type constructor, (4) an automatic persistence mechanism, (5) the possibility of separating classes and method definitions from the implementation and (6) last but not least, a better merge between the O_2 syntax and the host language ones, i.e. every implementation of O_2 on a given host language follows the syntax of the host language. The current prototype runs on Sun and implements all of the functionalities listed above. The $BasicO_2$ compiler is under implementation.

10 Acknowledgments

The authors thank F. Bancilhon and P. Kanellakis for their careful reading and comments which greatly improve the quality of this paper. The O_2 compiler was implemented by the language team of the GIP Altaïr which includes D. Excoffier, L. Haux. C. Lécluse and P. Richard. The module which is in charge of storing and managing classes and methods was designed and implemented by S. Gamerman and C. Delcourt. The object manager has been implemented by the system team. Numerous suggestions and improvements on the language and its syntax have been proposed by the Altaïr team and in particular by F. Bancilhon.

References

[AB 87] S. Abiteboul and C. Beeri, "On the Power of Languages for Manipulating Complex Objects", *International Workshop on Theory and Applications of Nested Relations and Complex Objects, Darmstadt, 1987.*

[AK 89] S. Abiteboul and P. Kanellakis, " Object Identity As A Query Language Primitive", internal report, 1989.

[AN 86] H. Ait-Kaci and R. Nasr, "LOGIN: A Logic Programming Language with Built-in Inheritance", *Journal of Logic Programming, 1986.*

[ACO 85] A. Albano, L. Cardelli and R. Orsini, "Galileo: a Strongly typed, Interactive Conceptual Language", *in ACM Trans. on Database Systems, 10(2):230-260, June 1985.*

[AH 87] T. Andrews and C. Harris, "Combining language and Database Advances in an Object-Oriented Development Environment", *Proc of the OOPSLA Conference, October 1987.*

[AtB 87] M.P. Atkinson and O.P. Buneman, "Types and Persistence in Database Programming Languages", *ACM Computing Surveys, June 1987.*

[BK 86] F. Bancilhon and S. Khoshafian, "A Calculus for Complex Objects", *ACM PODS, 1986.*

[Ban et al 88] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lécluse, P. Pfeffer, P. Richard and F. Velez, "The design and Implementation of O_2, an Object-Oriented Database System", *in Advances in Object-Oriented Database Systems Springer-Verlag, September 1988.*

[Ban 88] F. Bancilhon, "Object-Oriented Database Systems", *Proc of PODS 88, Austin, March, 1988.*

[Ban et al 87] Banerjee J., et al., "Data Model Issues for Object-Oriented Applications", *ACM trans. on Office Information Systems, Jan 1987.*

[Ber et al 88] P. Bernstein et al., "Future directions in DBMS Research", *Workshop of the International Computer Science Institute, Feb 4-5, 1988.*

[Car 84] L. Cardelli, "A Semantics of Multiple Inheritance", Semantics of Data Types, *Lecture Notes in Computer Science, 1984.*

[CW 85] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", *ACM Computing Surveys, Vol 17:4, 1985.*

[CDV 88] M. Carey, D. DeWitt and S. Vandenberg, "A Data Model and Query Language for EXODUS", *Proc of the ACM-SIGMOD Conference, Chicago, 1988.*

[Car 87] M. Caruso, "The VISION Object Oriented Database Management System", *Proc of the Workshop on Database Programming Languages, Roscoff, France, Sept. 1987.*

[DKV 87] S. Danforth, S. Khoshafian and P. Valduriez, "FAD - A Database Programming Language", *MCC Technical Report, October, 1987.*

[DFKLR 86] N. P. Derrett, D. H. Fishman, W. Kent, P. Lyngbaek and T. A. Ryan, "An Object-Oriented Approach to Data Management", *Compcon 31 IEEE Computer Soc. Int. Conference, 1986.*

[DGL 87] K. Dittrich, W. Gotthard and P. Lockeman, "DAMOKLES - The Database System for the UNIBASE Software Engineering Environment", *Database Engineering, Vol 10, No 1, March 1987, pp 37-47.*

[GR 83] A. Goldberg and D. Robson, "Smalltalk80: The Language and its Implementation", *Addison Wesley, 1983.*

[KBCGW 87] W. Kim, J. Banerjee, H-T. Chou, J. F. Garza and D. Woelk, "Composite Object Support in an Object-Oriented Database System", *Proc of the OOPSLA Conference, October 1987.*

[Kup 85] G. Kuper, "The Logical Data Model: a new Approach to Database Logic", PhD thesis, Standford University, September 1985.

[LRV 88] C. Lécluse, P. Richard and F. Velez, "O_2, an Object-Oriented Data Model", *Proc of the ACM-SIGMOD Conference, Chicago, 1988.*

[LR 88] C. Lécluse, P. Richard, "Modeling Inheritance and Genericity in Object-Oriented Databases", *Proc of the ICDT 88 Conference, Brugge, Aug 31-Sep 2, 1988.*

[LR 89] C. Lécluse and P. Richard. "Modeling Complex Structures in Object-Oriented Databases", *to appear in proc of the PODS 89 Conference, Philadelphia, March 29-31, 1989.*

[MOP 85] D. Maier, A. Otis and A. Purdy, "Development of an Object-Oriented DBMS", in *a Quaterly Bulletin of the IEEE Computer Society Technical Committee on Database Engeneering, Special issue on Object-Oriented Systems, Vol 8:4, 1985.*

[Pri 84] D. Price, "Introduction to ADA", *Prentice-Hall , Inc., Englewood Cliffs, new Jersey, 1984.*

[SCBKW 86] G. Schaffert, T. Cooper, B. Bullis, M. Kilian and C. Wilpot, "An Introduction to Trellis/Owl", *in Proc of the OOPSLA Conference, Portland, 1986.*

[Sch 81] D. W. Shipman, "The Functional data Model and the Data language DAPLEX", *ACM Transactions on Database Systems 6(1), pp 140-173, March 1981.*

[Str 86] B. Stroustrup, "The C++ Programming Language", *Addison Wesley, 1986.*

[VBD 89] F. Velez, G. Bernard and V. Darnis, "The O_2 object Manager, An Overview", Altaïr Internal Report, 1989.