An Active Meta-model for Knowledge Evolution in an Object-oriented Database

Z. BELLAHSENE
LIRMM
UMR 9928 CNRS - Montpellier II
161 rue ADA 34392 Montpellier Cedex 5
France
e-mail : bella@lirmm.fr

Abstract. It is not reasonable to consider that a single data model can be adequate for any application.There are many data models; each of them has its advantages and drawbacks. Building up an other model does not make sense in the actual state of research in the database field.
Our approach consists of a proposal of a meta-model providing an open environment to allow knowledge evolution in object-oriented database systems. Knowledge evolution means updates on database schema: propagation of updates from schema to instances and dynamic propagation of views update operations. Furthermore this meta-model enables the extension of data model concepts by modifying their semantics. By modifying and extending these model, it can be tailored or customised to suit various application domains.

Keywords : Object Oriented database, inheritance, meta-model, schema evolution, active database,...

1. Introduction :

Object-oriented database technology is coming to maturity; many Object-Oriented database Management Systems (OODBMS) are commercialised (Gemstone, ORIONII, O2, ONTOS,...). They have acquired better productivity and program maintainability and advances in the user interface area from object-oriented languages and software engineering. The most significant feature of object-oriented database systems is their solution for the impedance mismatch, provided by encapsulation which embodies data and programs(i.e methods) in the same object. The ability of OODBMS to build application systems in new fields such as CAD, CAI, multi-media databases and geographic information systems, hides some limitations due to a lack of flexibility. This concerns schema evolution which is a very important feature in such applications because of the manipulated objects own permanent dynamic nature. Most of OODBMS provide schema evolution facilities, but they seldom support automatic propagation to the object instances. Furthermore, these systems have not been developed with extensibility as a key goal.

While the notion of meta-level knowledge is well known in AI, very little literature on it has produced in the database field. The main idea is to view every knowledge representation in the system as an extended data type and write explicit description of each of them. Our approach provides a meta-level and to allow a specialized user the modification of its own data model to fit with specific application domain.
The primary focus of this work is to provide data model extensibility through the explicit representation of their semantics. Furthermore, the major features of this meta-model are :

- Performing automatic propagation of updates from schema to instances,
- Performing automatic propagation of view update operations,
- Allowing data model semantic concept modification.

In this paper, the problem of schema evolution has been studied according to a modelling point of view rather than a software one. Our approach is more flexible than the software one because it is easier to add to or modify representations and operations of "meta--objects" than of programs. Our meta-model integrates the Event-Condition-Action mechanism [8] to take into account the dynamic aspect of the meta-model and therefore deals with the automatic propagation of update operations to the database schema and its instances. For instance, in the schema evolution context, it enables declarative description of the legal change operations on a schema and on the corresponding instances. When one of these change operations is executed the system must trigger the update on instances without user intervention.

The remainder of this paper is organised as follows. Section 2 describes the study context : the object-oriented features. We give an overview of our meta-model in section 3. In section 4, we show how the meta-model can be used in the schema evolution context. In section 5, we study the implementation of the propagation of update operations to views. A summary and discussion of future research are provided in the last section.

2. The object context

In this section, we review the principle concepts of the object-oriented data models [1], [3], . . .

Object

The concept object is akin to that of entity when it covers semantic aspects. An object is defined through the set of attributes that characterize it and the operations it supports. In the field of database, the concept of "object" allows one to model the properties and the behaviour of an entity. An object has an identity that is independent of its value. This implies that the equality and identity operators on objects are different [7].

Class

The concept of class allows the grouping together of objects which have the same data structure (attributes) and the same operations called methods. A class is generally described by names, its attributes and methods. Classes are organised in a hierarchy of inheritance.
Example: here is a class defining a *Person* in DDL of the O2 system that we use in our implementation [12].

```
class Person
type tuple ( name : string,
             birth_date: Date,
```

```
            address: tuple ( road : string,  city: string, zip-code : string))
    method
          age: integer
    end;
```

Inheritance

Inheritance is a mechanism that allows the factorization of the parts common to several classes. Regarding modelling, the inheritance enables one to refine the definitions of object classes by introducing a specialization/generalization link. There are several types of inheritance, the best known are : simple and multiple inheritance. The former corresponds to a class hierarchy : each class that is at the ith level of the inheritance tree inherits the attributes and methods of the parent-class at the (i-1)th level (called superclass). In the case of multiple inheritance, classes are arranged in a graph (without cycle) : a class can then inherit from several superclasses. However, when the mechanism of multiple inheritance is applied, a conflict of inheritance may arise when the classes involved contain attributes or methods that have the same name. Several strategies to solve this kind of conflict have been suggested [10], [15].

3. The meta-model

In this section we present the basic concepts of our meta-model.We emphasize here the active and evolutive aspects of the meta-objects. Meta-data in database systems includes information on schema, constraints and view definitions. Our meta-model is designed to model and manage the data model concepts. The following figure represents a database architecture with three levels of abstraction. Each of them can be viewed as an instance of its upper level.

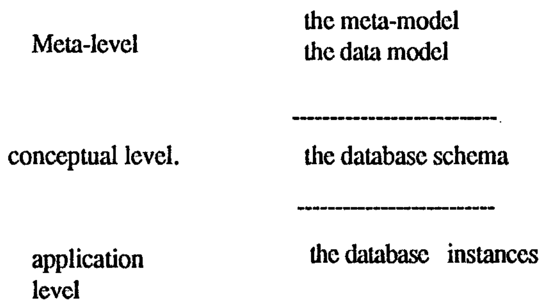


Fig 1. Levels of abstraction in a Database

3.1 The meta-model concepts

Its principles are based on the meta-class structure and the explicit representation of typed relations that link data model concepts. Furthermore the considered relations are types of dependency relations. Also, the meta-model integrates the ECA mechanism [5] to make the meta-objects active. The ECA rules will allow declarative description of the activation

condition by specifing events and actions to be executed in order to preserve the coherence of the dependency relations and to implement update propagation.

The primary concepts of our meta-model are :

.The *E-meta-class*, which can model every data model concept (for example class, instance,...)

.The *R-meta-class* is dedicated to representing the inter-concept links : aggregation, generalisation, inheritance and the following typed relations :

-class_instance (that links a class to its instances),
-instance_class (is the inverse of the former link),
-view_class (existing between a view and classes on which it is defined)
-class_view (link between a class and the views defined on it)

The meta-schema is a generalization hierarchy that indicates the global organization of the meta-model and what categories of concepts exists in the data model and the relationships between them. It makes extensive use of the concept of inheritance in the role of inter concept link. All the meta-objects are described by meta-classes while the inter-concept link is modelled by an R-meta-class. The following figure describes the meta-schema of our meta-model. The root of the meta-classes hierarchy is "OBJECT" and means that all concepts are objects in the meta-model. However there are two types of objects : the predefined objects (integer type, float type, string,...) and the abstract objects which are user-defined objects or concepts of the data-model. The meta-class "class" enables representation of the high level constructs of a data model, for instance the concept of class of the object-oriented data model or the relation concept of the relational data model.

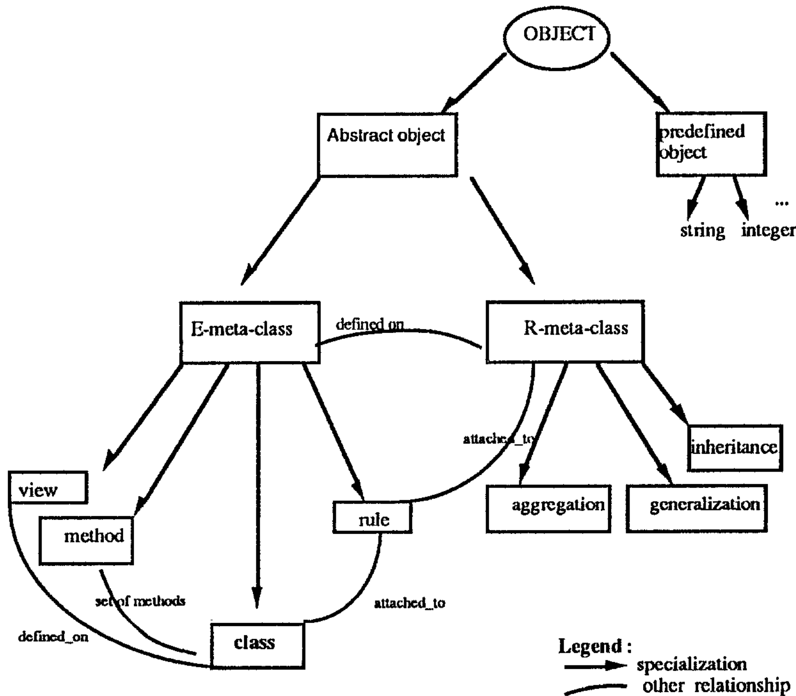


Fig. 2. The meta -schema

3.2 Extensibility of a data model:

Extensibility consists of the capability to add to a data model new types of classes and relationships or to extend the existing classes or relationships. The extensibility is achieved by modifying the meta-level concept specification. For example, the deletion of the meta-class "method" has as a consequence the change of the semantics of the data model and means that the data model is not object-oriented.

Class extensibility
Class extensibility is an extensibility whereby a data model can be extended to provide new types of classes: extending an existing class or adding a new class. This extensibility is achieved by modifying the meta-schema. For example, if we have to add a new class type called Z-class, we have only to create a "Z-object" meta-class as a sub-class of the meta-class "OBJECT" and to add "Z-class" as sub-class of the meta-class "class". Extending existing classes consists of sub-classing the meta-classes corresponding to such existing class types.

Relationship extensibility
This type of extensibility encompasses data model extensions such as various forms of inheritance and customised relationships to fit a particular domain by sub-classing the R-meta-classes representing such relationships. For example, to add a new type of inheritance, we have to define a sub-class of the R-meta-class called "inheritance"(see figure 2).

3.3 The active aspect

The main idea is to consider that the inter-concept relationships in the schema evolution modelling are dependency relationships.The two types of meta-objects involved in this kind of relationship are called influential meta-object and dependent meta-object.

The principle of a dependency relationship: When the influential meta-object is modified then the dependent meta-object must be updated in order to maintain the coherence dependency constraint.

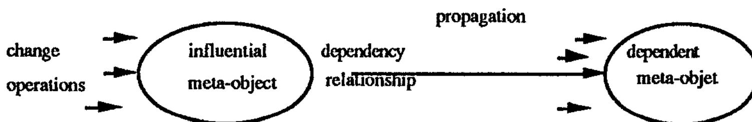


Fig. 3. dependency relationship

Now the problem is to represent these changes and their propagation to the dependent meta-objects.

ECA rules
The aim of this section is not to provide an exhaustive review of active database systems but to introduce the framework of our approach. Research in this area is conducted by the HiPAC project [DAYA 88] and by the Postgres database system project [11]. Active database systems attempt to provide modularity and timely response to critical situations : "the system is able to monitor the situations, trigger the related actions when the conditions are true and to schedule tasks to meet timing requirements without user intervention" [8].

The development of active database systems requires solutions for knowledge representation, execution, scheduling and performance problems. The issues of these areas are considered in the HiPAC project and are not discussed in this paper.Event-Condition-Action rules (ECA), are a central concept in HiPAC.They provide more flexibility than simple triggers by specifying events that activate the constraint checking and the actions to be performed if some constraints are violated.

ECA rules are implemented as objects. An ECA rule consists of the following elements:
Event : the role of the event part of an ECA rule is to activate the condition part. It describes temporal events, database operations (data definition and data handling, transaction control) or signals from processes.
Condition : a condition is defined as a set of queries that are evaluated when the rule is fired.
E-C coupling mode specifies when a condition is evaluated according to the transaction in which the trigger has been reported.
C-A coupling mode specifies when an action is done according to the transaction in which the condition is evaluated.
Action : is a sequence of operations to be executed when the rule is fired and thecondition is satisfied.

The dynamic aspect of our meta-model lies in the R-meta-class activation points, which are implemented as ECA rules.
The R-meta-class structure (representing a dependency relationship between two meta-classes) is specified informally as follows. It is composed by:
- context : (influential_object : meta-class_type, dependent_object : meta-class_type)
- Activation Point =
    - E : event that activates the condition (update database operations)
    - C: condition that must be satisfied
    - A : operations or program to execute when the conditions are true

The context part defines the set of the meta-objects that are linked by this dependency relationship. When the events corresponding to change operations are activated and if related conditions are true then the attached actions are executed to achieve propagation to the dependent object.

Here is the specification of an R-meta-class in Data Description Language of O2 DBMS [12], according to the meta-schema defined in the figure 4. This implementation is inspired from Smalltalk. From each meta-object, a list of dependent meta--objects can be accessed via the dependent link (i.e the attribute private_dependents).

```
class R_meta_class
        type tuple (private _dependents: unique set (Object1))
        method
```

```
        public addDependent (anObj: Object1),
        public dependents: unique set (Object1),
        public releaseall,
end;
```

4. A proposal for modelling schema evolution

In this section we first review some schema evolution possibilities provided by some object-oriented database systems. Then we will present our proposal for modelling the change operations on schema and their automatic propagation.

4.1 Schema evolution in object-oriented database systems

This sub-section is not a comprehensive study of existing research efforts but only a limited survey of the ability of some OODBMS to support the changes on schema and their propagation on the object instances. No system provides a full support for object evolution; most of them support changes in the object definitions. However, few of them have the ability to propagate these changes to the related instances. Because our approach is complementary to other works done in this domain, we will not make comparisons with them [2],[9],...

Structural consistency of a schema
In object-oriented database systems there are two basic types of consistency : namely, structural and behavioral consistency. The former one refers to the static part of a database. The behavioral consistency refers to the dynamic part of a database. It ensures that methods perform the "desired" task [13]. Many object-oriented database systems (ORION, GEMSTONE) do not support any mechanism for dealing with behavioral consistency.
Change operations have to ensure that a structurally-consistent schema is produced as a result of the update operation. Structural consistency is provided by using a set of "invariants" that define the consistency requirements of the class hierarchy. The main "invariants" which stemmed from the ORION database system and are now used in most OODBMS [2], are :

. *Class lattice invariant:* The sub-class/superclass relationship forms a lattice having as root the predefined class "Object".
Distinct name invariant: All instance variables or methods defined or inherited must have distinct names.
. *Distinct identity (origin) invariant:* All instance variables or methods defined or inherited must have a distinct origin.
. *Full inheritance invariant:* A class inherits all instance variables and methods from each of its superclasses unless it defines an instance variable or method with the same name.
. *The compatibility type invariant:* If an instance variable V2 of a class C is inherited from an instance V1 of a superclass of C then the type (or domain) of V2 is either the same as that of V1 or a subclass of V1.

These invariants must be preserved by any change on the schema.

Legal schema update operations :
An update operation on a schema is qualified as legal if and only if it ensures production of a consistent schema. Update operations on a database schema can be classified in two categories:

1) Changes to class definitions including :
- add an attribute or method to a class
- delete an existing attribute or method
- change the name of an attribute or the name of a method.
- change the type of attribute
- change the signature of a method or its code

2) Modifying the graph of classes
- add a new class to the graph
- delete an existing class and its links
- change the name of a class
- moving a class in the graph

Propagation of update operations
The problem of schema evolution cannot be limited to a set of change operations on the class definitions. The system must also provide capability to control update propagation on the object instances. There are two ways to realise update propagation : in fully automatic (ORION) or manual (ENCORE) mode.
When propagation is automatic, the delay of effective change propagation to the object instances has to be defined. Propagation can be performed immediately or can be deferred. The first mode emphasizes consistency and information preservation, whereas the second one allows the database to be available and makes the modification effective at the next access to the instance.

4.2 Modelling the schema evolution

Most of the OODBMS's provide schema evolution facilities. But they seldom support automatic propagation to the object instances. Our approach attempts to avoid this drawback by providing the means to model the schema legal update operations and their propagation to the associated object instances.The following figure describes the meta-schema; The stippled areas represent the meta-classes that are used to implement the schema evolution.

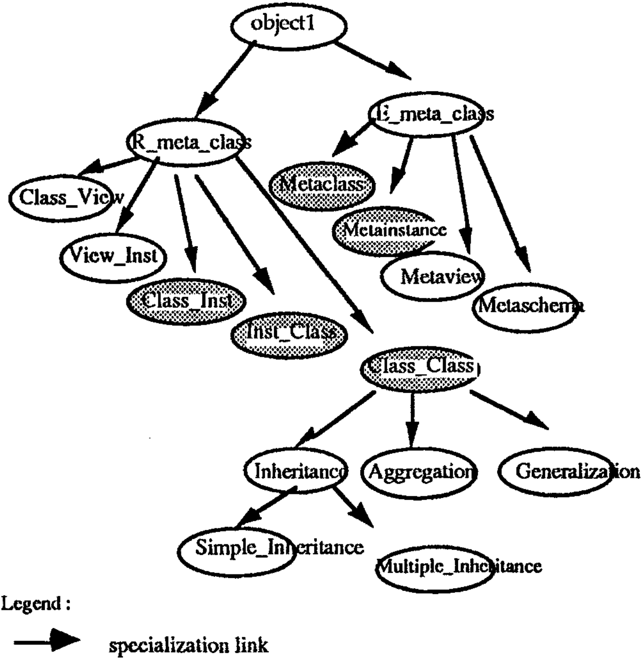


Fig. 4. The meta-schema for the schema evolution

. The E-meta-class called *"Metaclass"* describes the classes. It contains the following informations about each class : its class name, its superclassses, its attributes, its dependants links (noted ci) with its instances and its dependant links with other classes (noted cc). This implementation is inspired from Smalltalk. From each meta-object, a list of dependent meta--objects can be accessed via the dependent link modelled by a R_meta_class (i.e the attribues ci and cc).

The methods of *Metaclass* are update operations on the classes graph. When they are executed these methods trigger propagation operations on the schema or on the instances (which are implemented as methods in R-meta-classes) .The implementation of this meta-class, in DDL of the O2 system is :

```
class Metaclass
  inherit E_meta_class
  public type tuple ( class_name:string,
                      superclasses:unique set(Metaclass),
                      attributes:unique set(Attributes),
                      ci:Class_instance ,
                      cc:Class_view)
  method
        public init(class=name: string),
      /* schema update operations */
        public add_attribute (n:string, t: string),
        public del_attribute(name:string),
        public rename_attribute(old_name:string,new_name:string,
                                mc:Metaclass),
```

```
            public  del_class(option:string,mc:Metaclass)
    end;
```

.The E-meta-class "*Metainstance*" allows direct access to the instances of a class. It includes all the update operations on instances.
.The R-meta-class "*Class-class*" describes the inter-class relationships of aggregation or generalisation. Its methods model propagation of update operations to the definition class (addition, deletion of an attribute, ...). They are triggered by the execution of methods in the E-meta-class "Metaclass".
.The R-meta-class "*class-Inst*" represents the relationship existing between a class and its instances. It includes methods performing the propagation of update operations of class definition to instances. These methods are triggered by the execution of class definition operations implemented in the "Metaclass".

```
Class Class-instance
  inherit R-meta-class
  method public add_attribute(a: Attribute, mc : Metaclass),
         public del_attribute(name:string, mc:Metaclass)
end;
```

.The R-meta-class "*Inst-class*" is the inverse of the preceding relationship. Its methods perform propagation of the update operation of instances to the corresponding class. They are triggered by the execution of methods of the "Metainstance".

Example : Here is an concrete example of schema evolution

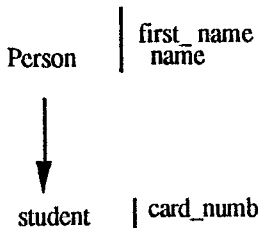


Fig. 5 An example of schema

This example shows a modelling of an operation that alters a schema definition by adding an attribute in a class. The implementation of this update operation uses the R-meta-class "class-instance" and the E-meta-class "Metaclass".

1) Creation of the *Person* class

```
class Person
        public type tuple ( name:string, first_name:string)
  end;

  Person_class = new Metaclass("Person");
```

The method new of "Metaclass" creates an instance of type Metaclass and calls the method init to initialize this instance. This method triggers the method add_class of the R-meta-class " Class_Class" to add this class (*Person*) in the schema.

2) Adding the attribute age in the *Person* class

```
  >Person_class->add_attribute("age","integer");
```

The method add_attribute triggers first the add_attribute("age","integer",self) of the R-meta-class "Class_Class": this latter modifies the *Person* class and its sub-class student by adding the attribute "age". And after that, the method add-attribute of Person class triggers the add_attribute("age","integer",self) of the "Class_Inst" R-meta-class : this latter adds the attribute "age" in all instances of the *Person* class and *Student* class. The value of this attribute is initialized to NULL.

4.3 Instance evolution and its propagation on the schema

Some research works deal with the problem of schema evolution and the propagation to the instance level but few of them have dealt with the problem of schema evolution depending on instance evolution. This fact reveals the rigidity of the DBMS where the instances must conform to their schema. Our meta-model enables representation and performing of instance evolution and automatic propagation to the schema. This can be done by describing the relationship that links instances to their class and by including in R-meta-class specifications, events, conditions on the propagation process and actions. For example, filling threshold can be defined for classes. These thresholds can be computed on whole instances or parts of them (attributes). The following examples of instance evolution show that our approach is relevant and powerful enough to support many knowledge evolution criteria.

Here are two examples of typed evolution of the schema corresponding to semantic relationships (generalization, specialization).

Generalization :
The schema evolution in this case consists of creating a new class and a generalization link. In the following example, instances evolution involves a generalization link between classes. It shows how an evolution process based on values of some attributes enables elimination of null values.

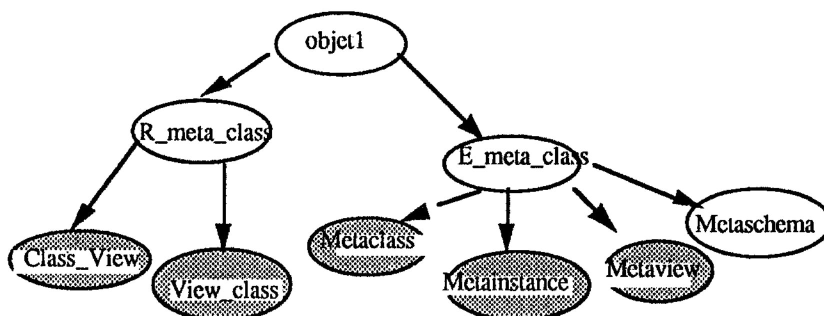
For instance : the class that represents "Students" and possesses an attribute CV for "Curriculum Vitae". When the amount of null values for attribute CV reaches 50%, a new class is created that is a generalization of the former without attribute CV. All instances representing students who have no Curriculum Vitae are attached to this class. This can be represented as follows :

R-meta-class : instance_class
  Context : (I : type_instance_meta-class, C : type_classe_meta-class,)
  E : insert instance operation
  C : 50 % of instances without CV
  A : create a super-class and group all instances without CV
End;

Specialisation :
This typed evolution of the schema involves the creation of a sub-class of a given class. The following example shows an evolution of the schema according to its instances evolution. The criterion used to trigger the propagation of this evolution is : "If 25 % of instances belonging to generic class cannot be attached to existing sub-class, a sub-class must be created to link these instances". It will be modelled by an R-meta-class that represents the dependent link (instance_of) between instances and their class.

5. Modelling evolution of views

Evolution of views represents one form of schema evolution [6]. In this section, we will attempt to show how our meta-model can be applied to implement the view update operations and their automatic propagation. A novel solution to the problem of view definition and updating has been proposed in [4]. The update operations on views include those of their schema evolution. However, only operations on the view schema which do not alter the global schema are allowed.
The following figure introduces the different R-meta-classes and E-meta-classes used to model the view evolution.



Legend:
——>specialisation link

Fig. 6. The meta-schema of views evolution.

The "*Metaview*" is the E-meta-class modelling the views. Its methods represent the update operations on views. The execution of these methods triggers the appropriate propagation to the instances or the classes. The following decription is conform to the view definition proposed in [4].
The specification of the E-meta-class "Metaview" in DDL of O2 system is :

```
class Metaview
  inherit E-meta_class
  public type tuple (view_name:string,
                     cpdv:Metaclass,
                     attributes:unique set(label),
                     condition:string)
  method public add_attribute (attr_name:string),
         public del_attribute (attr_name:string),
         public add_condition (c:string),
         public change_condition (c:string),
         public command(c:string)
end;
```

The R-meta-classes modelling the relationships between a view and classes are:
- "*Class-view*" describing the relationship between a class and the views defined on

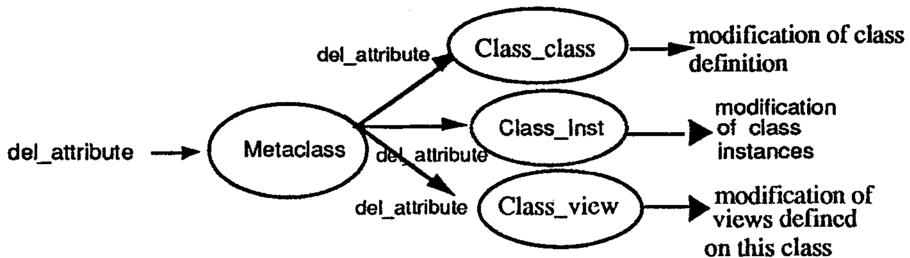
it. Its methods perform propagation operations of update on classes, to views. These methods are triggered by the execution of methods in "Metaclass".

- "*View-class*" represents the relationship between a view and the classes on which it is defined. Its methods perform the propagation of update operations on views, to the corresponding instances.

Activation of propagation of update operations

The execution of an update operation on the database activates an appropriate method in the associated E-meta-class : "Metaclass" or "Metaview" or "Metainstance". This method will trigger the corresponding methods in the R-meta-classes (which perform the update propagation) : "Class-class" or "class-instance" or "instance-class" or "Class-View" or "View_class".

Example: There is an example of an update operation on the global schema(the deletion of an attribute), named del-attribute. The execution of this operation activates another method in the E-meta-class "Metaclass" which in turn triggers appropriate propagation operations on the R-meta-classes "class-class", "class-instance" and "class-view".



Legend:
---> activation

Fig. 7 An example of view update operation

The specification of the R-meta-class "class-view" in DDL of O2 system is :

```
class Class_View
  inherit R_meta_class
  type tuple (private _dependents:unique set(Metaview),
  method
          public add_view (mv:Metaview),
          public del_attribute(name:string, mc:Metaclass),
          public rename_attribute (old_name, new_name:string, mc:Metaclass),
          public del_class(option:string, mc:Metaclass)
end;
```

6. Concluding Remarks

We have presented in this paper a meta-model that provides a novel solution to the problem of knowledge evolution in object-oriented database systems. The major purpose of this work is to provide data model extensibility through the explicit representation of their semantics. Our approach consists of proposing a meta-model that allows the

modification of the concept's semantic definition. The meta-model presented in this paper emphasizes the following objectives :

- dynamic propagation of schema change operations
- automatic propagation of views update operations, on the database.
- Allowing data model semantic concept modification and the integratioin of new concepts and relationships.

This approach requires an explicit representation of the data model concepts and the integration of an active mechanism. Therefore our meta-model is based on the explicit description of inter-meta-class relationships combining the ECA mechanisms in order to support the dynamic aspect of knowledge. The approach of modelling the propagation change operations is more generic and modular than the software one. By the meta-model, the propagation conditions and the actions to realise these propagations are specified in a declarative way, the system may then trigger the actions to be performed when the conditions are satisfied without user intervention.
A first version of our meta-model has been implemented with the O2 DBMS. This version includes :

- Automatic propagation of schema change operations on the object instances.
- View definition and automatic propagation of view update operations

Much remains to be done, in particulary concerning the extensibility aspect.
Another direction of research consists to apply the meta-model to deal with versions. Besides, our meta-model seems relevant to the representation of semantic heterogeneity in a multidatabase system. The explicit representation of information on each local schema, the views and on the global and the translation rules between them, will support the ability to reason about the semantic heterogeneity (for instance to ensure the view update propagation).

References

1. ATKINSON et al., "The Object-Oriented Database System Manifesto", in Proc. of first Conference on Deductive and Object-Oriented Databases, Kyoto, Japan,December 1989.

2. BANERJEE J., KIM W., KIM K.J., KORTH H., "Semantics and Implementation of Schemes Evolution in Object-oriented databases", in Proc. ACM SIGMOD Conference, San Francisco, May 1987.

3. BEERI C., "Formal models for Object Oriented Databases", Proc.DOOD89, Kyoto, Japan, Dec. 1989.

4. BELLAHSENE Z., "Vues et Points de Vue dans une Base de Données Orientée Objet", Technical Report, LIRMM, Juny 1992.

5. DAYAL U. et al., "HiPAC : Project Combining Active Databases and Timed-Constraints", in SIGMOD RECORD, Vol. 17, N°1, March 1988.

6. BERTINO E., " A View Mechanism for Object-oriented Database", 3rd International Conference on Extending Database Technologie, March 23-24, Vienna (Austria), 1992.

7. LECLUSE C., RICHARD P., VELEZ F., "O_2, An Object Oriented Data Model", Altair Technical Report 10-87, 15 Sep. 87.

8. MC CARTHY D. R. and DAYAL U., "The Architecture of An Active Data Base Management System", in Proc. ACM SIGMOD 89 Conference, Portland, Oregon.

9. NGUYEN gian Toan, RIEU Dominique, "Schema Evolution in Object-oriented Database systems", Data&Knowledge Engineering, North Holland, vol4 Jully, 1989.

10. SNYDER A., "Encapsulation and Inheritance in Object-Oriented Programming Languages", OOPSLA'86, Portand, OREGON, 1986.

11. STONEBRAKER and al., " The Implementation of POSTGRES", IEEE Transactions on`Knowledge and Data Engineering, vol 2, N° 1, March 1990.

12. Reference Manual, O2 Technology, 1992.

13. ZICARI R., "Incomplete Information in Object-oriented Databases", SIGMOD RECORD, September 1990.

14. ZICARI R., "A framework for O2 Schema updates", Altair Technical Report 38-89, October 1989.

15. CARRE B. et GEIB J-M., "The Point of View Notion for multiple Inheritance", in Proc. ECOOP/OOPSLA 90, October 1990.