ADO: An Active Distributed Object-Oriented Database Model

Ling Feng  
*University of Tilburg*, ling@kub.nl

Allan Wong  
*Hong Kong Polytechnic University*, csalwong@comp.polyu.edu.hk

Follow this and additional works at: [http://aisel.aisnet.org/amcis2000](http://aisel.aisnet.org/amcis2000)

Recommended Citation

[http://aisel.aisnet.org/amcis2000/176](http://aisel.aisnet.org/amcis2000/176)

This material is brought to you by the Americas Conference on Information Systems (AMCIS) at AIS Electronic Library (AISeL). It has been accepted for inclusion in AMCIS 2000 Proceedings by an authorized administrator of AIS Electronic Library (AISeL). For more information, please contact elibrary@aisnet.org.
ADO: An Active Distributed Object-Oriented Database Model

Ling Feng, Tilburg University, InfoLab, PO Box 90153, 5000 LE Tilburg, The Netherlands, ling@kub.nl
Allan Wong, Dept. of Computing, Hong Kong Polytechnic University, China, csalwong@comp.polyu.edu.hk

Abstract

Object-oriented databases are emerging as a new-generation database technology for complex applications. In this paper, we present an active distributed object-oriented (ADO) database model, which can capture not only passive behaviors but also active behaviors of complex objects. Besides, the distribution nature of objects can be reflected from within the model as well. Based on the model, we have developed a prototype active distributed object-oriented database management system, and applied it to a housing property management application.

Keywords: Object-oriented database, object, active object, object distribution, and database model.

1 Introduction

Object-oriented database systems are emerging as the new-generation database technology especially for complex applications (e.g., CAD/CAM, CASE, GIS, AI, etc.) in terms of rich data types, efficient support for complex objects, and computing power. Usually, these non-standard application areas require timely responses to critical situations, sophisticated constraint management and adaptiveness to changing business policies. It is thus desirable for the next-generation database systems to be active in the sense that they are able to react automatically rather than passively to certain events by means of knowledge (e.g., triggers) stored in the databases [KR98]. In addition, the rapid proliferation of computer networks has enabled users to easily access a large number of data sources scattered around different sites, making the distributed database management more and more important in the new era.

In this paper, we present an active distributed object-oriented (ADO) database model for our distributed object-oriented database management system. In the model, both passive and active behaviors of objects can be explicitly captured into classes. Moreover, different from previous object-oriented database models, the distribution nature of objects across sites can also be reflected in the database model. Based on the ADO model, we have successfully developed a prototype active distributed object-oriented database management system, which can manage both distributed and active objects. This system can be used in a number of complex applications, such as engineering design and manufacturing, geographic information systems, knowledge-based systems, scientific and multimedia databases, etc.

2 Related Work

Database management systems (DBMSs) have been available for three decades, originating in the form of the hierarchical and network models. In 1970, the relational data model was introduced by Codd [Cod70]. It is based on a simple and uniform data structure - the relation, and view data as collections of records in these relations. The relational data model revolutionized the database field by separating logical data representation from physical implementations. Its inherent simplicity laid a solid theoretical foundation for relational DBMSs, and led to the development of powerful, non-procedural query language [Cod71, Cod72a, Cod72b, Cod79].

While the relational data model hides many implementational details, it is nonetheless closer to how the DBMS stores data than to how a user thinks about the underlying real-world enterprise. To bridge this gap, several semantic data models have been proposed to assist in the process of database design [Abr74, AM86, HM81]. Semantic models allow the user to come up with a good initial description of the data in an enterprise. Research in semantic modeling has articulated a number of constructs which provide mechanisms for representing structurally complex interrelations between data (e.g., ISA relationships), typically arising in the real-world [HK87, HK90]. A widely used semantic data model called the entity-relationship model can facilitate pictorially description of entities and the rela-
tionships amongst them [Che76].

More recently, inspired by the object-oriented paradigm from programming languages, database researchers turn their attention towards incorporating the behavioral aspect of data into database models, and develop object-oriented data models [KBC+88, Kim90b, Kim90a, LRV88, PS87, MD90, Fe90, ADM+89, HtH98]. Basically, an object-oriented data model is a set of object-oriented concepts for modeling data. It allows the explicit representation of object classes (or abstract data types). Each object in a database is identified by its surrogate rather than by its value. The methods (operations) which are encapsulated within objects can be inherited from one class to another [RBP+91, ZM90, Kim93, DD95].

Besides building a completely new object-oriented data model from the ground up, database researchers pursue other alternative approaches, including the development of object-oriented middleware to provide a object programming layer on top of relational database systems, and the extension of relational data model with built-in object capabilities to object-relational data models [CD96, CCN+99, Sto96, LOL98]. Different from object-oriented data models, object-relational data models start with the relational model and SQL query language. The attractiveness of this approach is that object capabilities can be added to industrially-proven database systems that already offer many valuable data management functions [CCN+99].

Nowadays, object-based data models have strongly enhanced database support for applications where complex objects play a central role [DHR96]. In the area of standards, SQL3 is moving in the direction to incorporate objects into relational data model. The Object Database Management Group (ODMG) has drafted object-oriented database standards for an object data language (ODL), an object query language (OQL), and a C++ programming interface for manipulating and querying object databases [Cat96].

3 The ADO Database Model

The constituent components of the ADO database are objects, classes and their partial relationships. Each object is instantiated from a class. The class defines the structure of an object, as well as its dynamic behaviors. All classes in the database form an inheritance hierarchy, where a class can inherit all the structures and behaviors from its superclasses. In addition, a class can reference other classes to become an aggregate class.

We start a formal description of the ADO database model with the following notations:

- A finite set of attribute names \( \mathcal{ATT} \).
- A countably infinite set of identifiers \( \mathcal{ID} \).

### 3.1 Values

**Definition 3.1** A value can be

1. an atomic value \( v \in \text{DOM} \); or
2. a constructional value \( v \), where
   - \( v = \{v_1, v_2, \ldots, v_m\} \), for \( j \leq j \leq m \), either
     - \( v_j \in \mathcal{ID} \) or \( v_j \in \text{dom}(\mathcal{D}_i) \) (1 \( \leq i \leq n) \), called a set value; or
   - \( v = \langle v_1, v_2, \ldots, v_m \rangle \), for \( j \leq j \leq m \), either
     - \( v_j \in \mathcal{ID} \) or \( v_j \in \text{dom}(\mathcal{D}_i) \) (1 \( \leq i \leq n) \), called a list value; or
   - \( v = [a_1 : v_1, \ldots, a_m : v_m] \), for \( j \leq j \leq m \), \( a_j \in \mathcal{ATT} \) and \( v_j \in \text{DOM} \cup \mathcal{ID} \), called a tuple value.

An empty value is the only value that can be used as an atomic value, a set value, a list value, or a tuple value, denoted as \( \phi \), \{ \}, \langle \rangle , or [ ] , respectively. Corresponding to atomic values, we call set, list and tuple values uniformly as constructional values. Let \( \mathcal{V} \) denote the set of all possible values in the database system, including both atomic and constructional values.

### 3.2 Objects

Any real-world entity is an object with a system-wide unique identifier. An object encapsulates a static state and dynamic behaviors. The values of attributes of an object constitute the state of the object, and the methods associated with the object operate on its state. Since all the objects of a class have the same methods, we delegate the behavioral capabilities of objects to their classes, and simplify the object definition as follows:

**Definition 3.2** An object \( o \) is a 2-tuple \( (o_{id}, o_{val}) \), consisting of an identifier \( o_{id} \) and a value \( o_{val} \), where \( o_{id} \in \mathcal{ID} \) and \( o_{val} \in \mathcal{V} \).

**Definition 3.3** Given two objects, \( o \) and \( o' \), they are

1. **identical**, if and only if \( o_{id} = o'_{id} \); and \( o_{val} = o'_{val} \).
2. **equivalent**, if and only if \( o_{val} = o'_{val} \).

The values of two identical objects are the same. In other words, if \( o_{id} = o'_{id} \), then \( o_{val} = o'_{val} \). However, \( o_{val} = o'_{val} \) does not imply \( o_{id} = o'_{id} \).

Assume function \( \text{ref}(o) \) return a set of identifiers, referenced within the value of object \( o \).
Definition 3.4 A set of objects $O$ is closed if and only if the following conditions hold:
1) $O$ is finite;
2) $\forall a, a' \in O$, $a_\text{val} \neq a'_\text{val}$;
3) $\forall o \in O$, $\text{ref}(o) \subseteq \{x_\text{id} \mid x \in O\}$. □

3.3 Structures

Values are instances of types, which can be either basic types (e.g., integer, float, char, boolean, etc.), or constructional types. We call the structure of a basic type a basic structure, and the structure of a constructional type a constructional structure. A constructional structure can be built using the set, list and tuple constructors. Let $B$ and $C$ denote the set of basic and constructional structures, respectively. The whole set of structures in the system $S = B \cup C$ can be recursively defined as follows:

Definition 3.5 A structure $s$ can be
1) a basic structure; or
2) a constructional structure, where
• $s = \{x\}$, where $x$ is a structure ($x \in S$), called set structure; or
• $s = \langle x \rangle$, where $x$ is a structure ($x \in S$), called list structure; or
• $s = [a_1 : x_1, \ldots, a_m : x_m]$, for $\forall j (1 \leq j \leq m)$, $a_j \in ATT$, and $x_j$ is a structure ($x_j \in S$), called tuple structure.

Definition 3.6 Given two tuple structures in $C$, $s = [a_1 : x_1, \ldots, a_m : x_m]$ and $s' = [a_1 : s_1, \ldots, a_m : s_m]$, $s$ is a sub-structure of $s'$, denoted as $s \leq s'$, if and only if $s$ has more attributes than $s'$, i.e., $m \leq n$. In other words, the sub-structure $s'$ of $s$ is more specific, while structure $s$ is more general. □

Assume function refer($s$) return the set of structures, referenced within the structure $s$.

Definition 3.7 A set of structures $\Delta$ is called a schema, if and only if the following conditions hold:
1) $\Delta$ is finite;
2) Each structure has a unique structure name;
3) $\forall s \in \Delta$, refer($s$) $\subseteq \Delta$. □

Definition 3.8 Let $\Delta$ be a schema and $O$ be a closed object set. An interpretation of structure $s \in \Delta$ on $O$ can be defined using a mapping function $I : \Delta \rightarrow O$ as follows:
1) when $s$ is a basic structure (assume $s$ is the structure of the basic type $D$), without loss of generality, then $I(s) = \{o \mid (o \in O) \land (o_\text{val} \in \text{dom}(D))\}$;
2) when $s$ is a constructional structure,
• if $s = \langle x \rangle$, then $I(s) = \{o \mid (o \in O) \land (o_\text{val} \in \text{set value}) \land (\text{for any set element } e_i \in o_\text{val}, e_i \in I(x))\}$;
• if $s = \{x\}$ (where $\forall o \in O$, $\text{ref}(o) \subseteq \{x_\text{id} \mid x \in O\}$), then $I(s) = \{o \mid (o \in O) \land (o_\text{val} \in \text{list value}) \land (\text{for any list element } e_i \in o_\text{val}, e_i \in I(x))\}$;
• if $s = [a_1 : x_1, \ldots, a_m : x_m]$ (where $\forall o \in O$, $\text{ref}(o) \subseteq \{x_\text{id} \mid x \in O\}$), then $I(s) = \{o \mid (o \in O) \land (o_\text{val} \in \text{tuple value}) \land (\text{for any attribute value } e_a \in o_\text{val}, e_a \in I(x))\}$.

Definition 3.9 Let $\Delta$ be a schema and $O$ be a closed object set.
1) An interpretation $I(s)$ of structure $s \in \Delta$ on $O$ is smaller than another interpretation $I'(s)$, if and only if $I(s) \subseteq I'(s)$.
2) The model of schema $\Delta$ on $O$ is the maximum interpretation $I^*$, where for $\forall s \in \Delta$, $I^*(s) = \text{Max}(I(s))$. □

Theorem 3.1 The model of a schema $\Delta$ on a closed object set $O$ always exists.
Proof 3.1 Since $O$ is a closed object set, according to Definition 3.4, there exists a finite set of interpretations on $O$. The theorem holds if we can prove that the union of two interpretations is also an interpretation, as the maximum interpretation is just the union of all the interpretations.

Without loss of generality, let $I$ and $I'$ be any two interpretations, and $I'' = I \cup I'$. For $\forall s \in \Delta$, $I''(s)$ satisfies the property 1) in Definition 3.8 when $s$ is a basic structure.

In the case that $s$ is a set structure ($s = \{x\}$, $x \in \Delta$), according to Definition 3.8, any element in $I'(s)$ or $I''(s)$ is in $I(x)$, thus, any element in $I''(s) = I(s) \cup I'(s)$ is also in $I(x)$, satisfying the property 2) in Definition 3.8. Similarly, we can prove the tenability of the property 2) for $I''(s)$ when $s$ is a list or tuple structure. □

3.4 Passive Behaviors

We use methods to describe the passive behaviors of objects. On receipt of a message consisting of a name and some arguments, the object will search the matching method, and then execute the corresponding program code of the method.

Definition 3.10 Let $\Delta$ be a schema. A signature on $\Delta$ is a mapping $s_1 \times \ldots \times s_n \rightarrow s$, where $s_1, \ldots, s_n, s \in \Delta$.

Definition 3.11 A method $m$ is a 3-tuple ($m_\text{name}$, $m_\text{sig}$, $m_\text{code}$), consisting of a method name $m_\text{name}$, a signature $m_\text{sig}$, and a program code $m_\text{code}$.
3.5 Active Behaviors

The active behaviors of objects are expressed by means of triggers, i.e., event/condition/action (ECA) rules [BM91]. When a certain event arises and the condition holds, the action will be triggered for execution.

**Definition 3.12** A trigger $g$ is a 5-tuple $(g_{\text{name}}, g_{\text{event}}, g_{\text{condition}}, g_{\text{action}}, g_{\text{time}})$, consisting of a trigger name $g_{\text{name}}$, an event in terms of a method $g_{\text{event}}$, a predicate $g_{\text{condition}}$ defined on attributes, a triggered method $g_{\text{action}}$, and the time $g_{\text{time}}$ to execute $g_{\text{action}}$. The time $g_{\text{time}}$ can be either immediately following the triggering method $g_{\text{event}}$, or before the commit of $g_{\text{event}}$.

3.6 Classes

A class is used to create a set of objects that share the same structure, and the same passive and active behaviors.

**Definition 3.13** Let $\Delta$ be a schema. A class $c$ is a 4-tuple $(c_{\text{name}}, c_{\text{structure}}, c_{\text{methods}}, c_{\text{triggers}})$, consisting of a class name $c_{\text{name}}$, a structure $c_{\text{structure}} \in \Delta$, a set of methods $c_{\text{methods}}$, and a set of triggers $c_{\text{triggers}}$ associated with class $c$.

We define two partial relationships (i.e., is-a and part-of) of classes as follows:

**Definition 3.14** Class $c$ is a subclass of class $c'$, denoted as $c \leq_{\text{is-a}} c'$, if and only if the following conditions hold:
1. $c_{\text{structure}} \leq_{\text{set}} c'_{\text{structure}}$;
2. $\forall m \in c_{\text{methods}}, \exists m' \in c'_{\text{methods}}$, such that $m_{\text{name}} = m'_{\text{name}}$;
3. $\forall m' \in c'_{\text{methods}}, \exists g \in c_{\text{triggers}}$, such that $g_{\text{name}} = g'_{\text{name}}$.

**Definition 3.15** Let $[a_1 : s_1, \ldots, a_n : s_n]$ be the structure of the class $c'$. Class $c$ is an aggregate class of $c'$, denoted as $c \leq_{\text{part-of}} c'$, if and only if $\forall s_k (1 \leq k \leq n) (s_k = c_{\text{structure}})$.

**Definition 3.16** Two classes $c$ and $c'$ are relevant, denoted as $c \leftrightarrow_{\text{rel}} c'$, if and only if each of the following conditions holds:
1. $c \leq_{\text{is-a}} c'$, or $c' \leq_{\text{is-a}} c$;
2. $c \leq_{\text{part-of}} c'$, or $c' \leq_{\text{part-of}} c$.

**Definition 3.17** A class lattice $\Gamma$ is a set of classes, if and only if the following conditions hold:
1. $\Gamma$ is finite;
2. Each class has a unique class name. That is, $\forall c, c' \in \Gamma, c_{\text{name}} \neq c'_{\text{name}}$;
3. $\forall c \in \Gamma$, $(c')_{\text{is-a}} c \subseteq \Gamma$;
4. From a unique root class $c_{\text{root}} \in \Gamma$, all other classes in $\Gamma - \{c_{\text{root}}\}$ can be traced. That is, for $\forall c \in \Gamma$, either $c \leftrightarrow_{\text{rel}} c_{\text{root}}$, or $\exists c_1, \ldots, c_n \in \Gamma$, such that $c \leftrightarrow_{\text{rel}} c_1 \leftrightarrow_{\text{rel}} \ldots \leftrightarrow_{\text{rel}} c_n \leftrightarrow_{\text{rel}} c_{\text{root}}$.

3.7 Distributed Databases

A database schema is a class lattice, whose instances are objects belonging to the classes in this lattice. The value of each such object, in terms of its attributes, matches the structure of the associated class. In other words, the object value must be within the maximum interpretation of the tuple structure of the class.

In our system, objects of a class can be horizontally partitioned based on either the class itself or its relevant class. The former self-class-based partition is similar to the traditional horizontal method used in relational databases, where partitioning is performed according to a predicate defined on attributes of this class. For the latter relevant-class-based partition, partitioning of a class arises from the fragmentation of its relevant classes. Thus, the partitioning predicate is defined on the attributes of its relevant class instead.

**Definition 3.18** Let $\Gamma$ denote a class lattice. A fragmentation $\Phi$ of class $c \in \Gamma$ is a tuple $(\Phi_{\text{rel}}, \Phi_{\text{node}}, \ldots, \Phi_{\text{node}_{\text{root}}})$, specifying the partitioning predicate $\Phi_{\text{rel}}$ and the nodes $\Phi_{\text{node}}$, ..., $\Phi_{\text{node}_{\text{root}}}$ to which the corresponding partition is distributed.

Assume function Object$(c)$ return the whole set of objects in class $c$, and function $\Phi_{\text{Object}}(c, f_i)$ return a set of objects in the fragment $f_i$ of class $c$.

**Definition 3.19** Let $\Gamma$ denote a class lattice. A fragmentation schema of class $c \in \Gamma$ is a tuple $\Lambda = (\Lambda_{\text{type}}, \Lambda_{\text{ref}}, \Lambda_{\text{f1}}, \ldots, \Lambda_{\text{fn}})$, where
1. $\Lambda_{\text{type}}$ is the partitioning method, which can be either “self-class-based” or “relevant-class-based”;
2. if $\Lambda_{\text{type}} = \text{"relevant-class-based"}$, then $\Lambda_{\text{ref}}$ is the relevant class where the partitioning predicates apply;
3. $\Lambda_{\text{f1}}, \ldots, \Lambda_{\text{fn}}$ is a list of fragments of class $c$, subject to the following constraints:
   - For $\forall o \in \Phi_{\text{Object}}(c, f_i) (1 \leq i \leq n)$, it satisfies the predicate of fragmentation $\Lambda_{fi}$ if $\Lambda_{\text{type}} = \text{"self-class-based"}$; Otherwise, its relevant class $o'$ (where $o' \in \Phi_{\text{Object}}(c', f_i)$ and $o \leftrightarrow_{\text{rel}} o'$) satisfies the predicate of fragmentation $\Lambda_{fi}$.
   - For $\forall o \in \text{Object}(c)$, there exists only one fragment $f_i (1 \leq i \leq n)$, such that $o \in \Phi_{\text{Object}}(c, f_i)$.
   - Object$(c) = \bigcup_{i=1}^{n} \Phi_{\text{Object}}(c, f_i)$
   - $\forall f_i, f_j \in \{f_1, \ldots, f_n\}$, \( \Phi_{\text{Object}}(c, f_i) \cap \Phi_{\text{Object}}(c, f_j) = \emptyset \)

**Definition 3.20** A database is a 4-tuple $DB = (NDB, \Gamma, O, T)$, consisting of a database name $NDB$, a class lattice $\Gamma$, a closed object set $O$, and a class fragmentation schema set $T$ in $\Gamma$, where for $\forall o \in O$, there exists a class $c \in \Gamma$, such that $o \in I^*(c_{\text{structure}})$.
4 A Housing Property Management Application

Based on the ADO database model, we have built up a housing property management application for a newly developed region. The core functions of the system include:

- effectively storing all the housing information, such as housing type (state, private, commercial), location, construction company, time of completion, architecture, number of rooms, decoration, etc., in the area.
- multidimensionally viewing all the transportation facilities, such as roads, railways, gas stations, etc., in the area.
- demonstrating all services, such as shops, hotels, etc., in the area.
- easily querying and managing housing information and surrounding facilities.

Figure 1: One portion of the application database schema

Each entity in the schema is represented by a class.

1) class-name: AREA
class-structure:
[  areaNo: integer;
  name: char[12];
  position: char[4];
  picture: IMAGE;
  sound: VOICE;
  transport: Struct(TRANSPORT);
  service: Struct(SERVICE);
  living: Struct(HOUSE);
  ......  ]
class-method:
void showPicture (IMAGE picture) {...}

void soundIntroduce (VOICE sound) {...}

......
class-trigger:
(  name: synBroadcast
   event: showPicture (IMAGE picture)
   condition: TRUE
   time: IMMEDIATE-AFTER
   action: soundIntroduce (VOICE sound)
  )

......
AREA is the class name. IMAGE and VOICE are two additional data types introduced for handling image and audio multimedia objects in the system. As their values are actually big sequences of characters, we view them as a special kind of basic data types. Inside the AREA class structure, the attributes transport, service, living get values from the structures of class TRANS-PORT, SERVICE, HOUSE, respectively. Two methods of the class AREA, showPicture and soundIntroduce, introduce the area in visual-audio ways. One trigger of the class AREA is named synBroadcast. Its active function is that once the showPicture method is called and executed, immediately following it, the soundIntroduce method will be automatically invoked.

2) class-name: TRANSPORT
class-structure:
[  trafficMap: IMAGE;
  road: ROAD-ST;
  railway: RAILWAY-ST;
  gasStation: STATION-ST;
  ......  ]
class-method:
void displayRoad (ROAD-ST road) {...}
void displayRail (RAILWAY-ST railway) {...}
void displayGasStation (STATION-ST gasStation) {...}

......
ROAD-ST, RAILWAY-ST, and STATION-ST are three constructional data structures, defined as follows:

ROAD-ST =
{  [roadName: char[12], roadPosi: { <float, float> } ] }.

RAILWAY-ST =
{  [railName: char[12], railPosi: { <float, float> } ] }.

STATION-ST =
{  [stationName: char[12], stationPosi: { <float, float> } ] }.

3) class-name: SERVICE
class-structure:
[  name: char[24];
  kind: char[12];
TEXT is another special basic data type introduced for handling strings of variable lengths.

4) class-name: HOTEL
superclass: SERVICE
class-structure:
  [  
    star: integer;  
    entertain: char[36];  
    breakfast: char[36];  
    diet: char[24];  
    facility: TEXT;  
    ......  
  ]

5) class-name: SHOP
superclass: SERVICE
class-structure:
  [  
    kind: char[24];  
    goodsOnSale: TEXT;  
    guide: TEXT;  
    ......  
  ]

6) class-name: HOUSE
class-structure:
  [  
    constructCompany: char[36];  
    completeTime: DATE;  
    architecture: char[12];  
    roomNumber: integer;  
    type: char[12];  
    decorate: TEXT;  
    location: char[24];  
    ......  
  ]

7) class-name: PRIVATE
superclass: HOUSE
class-structure:
  [  
    ownerName: char[24];  
    ownerAddr: char[36];  
    ......  
  ]

8) class-name: STATE
superclass: HOUSE
class-structure:
  [  
    organization: char[36];  
    purpose: TEXT;  
    ......  
  ]

9) class-name: COMMERCIAL
superclass: HOUSE
class-structure:
  [  
    price: float;  
    deal: BOOLEAN;  
    payMethod: char[24];  
    ......  
  ]

class-method:
  int DealStatistic (int price) {...}
  /* count the number of houses above a certain price that have been dealt. */

Due to space limitation, we illustrate the fragmentation design of class HOUSE and its subclasses, and omit the details of other classes’ fragmentation schemas. The objects of class HOUSE, STATE, PRIVATE, COMMERCIAL are distributed based on the following schemas.

5 Conclusion

In this paper, we present an active distributed object-oriented (ADO) database model, based on which a prototype active distributed object-oriented database management system has been developed. We have applied the system to a housing property management application, which can handle complex objects including texts, graphics, images, audio and video objects.

References


