Object Deputy Model and Its Applications

Yahiko KAMBAYASHI  Zhiyong PENG
Integrated Media Environment Experimental Laboratory
Faculty of Engineering, Kyoto University
{yahiko, peng}@kuis.kyoto-u.ac.jp

Abstract
In this paper, the concept of object deputy model is introduced to realize flexible objectbases. The model was required to realize advanced database applications, such as geographic databases, virtual office systems and distant education systems which are currently developed in our group. The model can also treat well-known problems of object-oriented data models: realization of flexible views, objects with more than one role, object migration history and multiple inheritance. An object appearing in a view can be regarded as playing a role, object migration corresponds to role change history of an object and multiple inheritance realizes aggregation of object roles, thus these problems can be handled uniformly. A deputy object has attributes and methods not in the source object. It has own object identifier, attribute values computed from attribute values of the source object, and methods generated from these of the source object. Furthermore, deputy classes are also introduced. The model realizes generalized inheritance mechanisms. A model for distributed databases and knowledge bases, data-knowledge coordination model, can be regarded as a model to realize very flexible objectbases. It is shown that the object deputy model can simulate data-knowledge coordination model. Applications to geographic databases, virtual office systems and distant education systems are also discussed.

1 Introduction
The data modeling capabilities of traditional (hierarchical, network and relational) are insufficient for development of advanced database applications, such as CAD, office automation and multimedia information processing. Object-oriented data model[20] is thought of as the most likely candidate to meet the complex data management requirements of advanced applications. However, the commercialized object-oriented databases(OODBs) display serious shortcomings in their flexibility and ability to model both the many-faceted nature and dynamic nature of real-world entities.

In order to overcome these difficulties, there are various kinds of researches on extension of object-oriented data model to add flexibility and advanced modeling capabilities, such as object views, roles, migration, multiple inheritance and data-knowledge coordination model. In this paper, we discuss such trends to specify requirements.

By comparing typical data models, we will show that separation of data property and application property can achieve flexibility. However, object-oriented data model follows the mixed approach like hierarchical and network data model, which can be used to realize high performance applications. Data-knowledge coordination model was introduced to realize a flexible multibase(databases, knowledge-bases and object-bases) system under distributed environment. It consists of three independent hierarchies, for (1) data property, (2) application property and (3) their dynamic coupling. Although the model can be used to realize flexible objectbases, its overhead cannot be accepted by various applications.

In order to achieve a reasonable tradeoff between efficiency and flexibility, we introduced a new data model called object deputy model that is an extension of object-oriented data model with the concepts of deputy objects and deputy classes. Deputy object is introduced to play a role instead of its source object. Its schema is defined by deputy class. An object can have multiple deputy objects which represent its many faceted nature. There is a bilateral link between an object and one of its deputy objects that can be used to realize data update propagation. The deputy objects have their own identifiers and may have additional attributes/methods, which make them have some independence. On the other hand, not all of attributes and methods of the source object can be inherited by its deputy objects. The inheritance is restricted by switching operations. Switching opera-
Table 1: Comparisons on Databases, Knowledge-bases and Objectbases

<table>
<thead>
<tr>
<th></th>
<th>Basic Contents</th>
<th>Property of Data</th>
<th>Property of Application</th>
<th>Typical Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Databases</td>
<td>Data</td>
<td>Semantic Constraints</td>
<td>Navigation Links</td>
<td>Network Model</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Schema Information)</td>
<td></td>
<td>Relational Model</td>
</tr>
<tr>
<td>Knowledge-bases</td>
<td>Data, Knowledge</td>
<td>Knowledge</td>
<td></td>
<td>Relational Model with Recursive Query Processing Capability</td>
</tr>
<tr>
<td>Objectbases</td>
<td>(Data-Methods)</td>
<td>Methods</td>
<td>Methods</td>
<td>Object-Oriented Model</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Extended Relational Model</td>
</tr>
</tbody>
</table>

Table 1: Comparisons on Databases, Knowledge-bases and Objectbases

2 Requirements for New Data Models

In this section, we will summarize data models by our standpoint, in order to show three major requirements for new data models.

Table 1 shows comparison of databases, knowledge bases and objectbases. These are classified by basic contents. In databases, basic contents are data, and property of data is expressed by database schema (functional dependencies by key constraints, join dependencies by sets of relations) as well as additional constraints (referential integrities, existence dependencies, triggers). Properties of applications are also expressed in network models as navigation links, while in relational schema it is not expressed. Knowledge base systems store both data and knowledge. Knowledge corresponds to rules to generate a set of data and semantic constraints on data. Various kinds of semantic constraints can be expressed. In knowledge base systems, data and knowledge are separately stored. In objectbase systems, however, data and methods are strongly combined, since each object is a combination of data and methods. Methods are used to express properties of both data and applications.

Due to the recent requirements on using databases to varieties of applications which share common data, knowledge and methods, it is required to develop a new data model which can be used for databases, knowledge bases and objectbases. One known approach is extension of relational model to meet such a requirement.

[Requirement I] Development of data models which can be used for databases, knowledge bases and objectbases.

Table 2 shows a comparison of typical data models based on efficiency, flexibility and capability. Especially the following two factors are important.

(1) Representation of data property
(2) Representation of application property

Although property of applications is usually expressed in each application program, part of them can be reflected in data schemata. There is a tradeoff problem between efficiency and flexibility. An efficient system can be realized if the both factors are combined together. On the other hand, a flexible system is realized by handling these factors separately. Although efficiency is the most important factor to manipulate large amount of data, flexibility is preferred by the following requirements.

a) A system is used for various applications.
b) Lifetime of data should be long.
Table 2: History of Data Models

<table>
<thead>
<tr>
<th>System</th>
<th>Contents</th>
<th>Property of Data</th>
<th>Property of Applications</th>
<th>User Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database Systems</td>
<td>Data</td>
<td>Hierarchical/Network Model</td>
<td>Relational Model</td>
<td>Easy to Use</td>
</tr>
<tr>
<td>Objectbase Systems</td>
<td>Data + Method</td>
<td>Hypertext/Hypertext</td>
<td>Object-Oriented Model</td>
<td></td>
</tr>
</tbody>
</table>

Characteristics of Systems

<table>
<thead>
<tr>
<th>Efficient</th>
<th>Flexible</th>
</tr>
</thead>
</table>

c) Integration of different systems under heterogeneous environment is required.

Systems based on hierarchical model and network model are efficient, since their structures reflect both semantic constraints of data (property of data) and navigation structures (property of applications). These are not efficient for applications whose properties are not reflected to the data structure.

In relational model, property of data is reflected to schema design (i.e., normalization of relations using data dependencies). As property of applications is not used for schema design, there is no preferred application. In order to achieve reasonable efficiency for each application, various kinds of query optimization techniques have been developed. Because of its flexibility, relational model is used as basis for deductive database systems, central model for integration of schemata in different models.

In most programming languages, there are the following two undecidable problems.

(a) Estimation of computation time (halting problem of Turing machines)

(b) Deciding equivalence of two programs.

Although relational language has less power than conventional programming languages (which correspond to Turing machine capability), it has the following nice properties:

(a) Any query expressed by relational language can be executed in time proportional to polynomial of the number of data (that is \( n \log n \)).

(b) Equivalence problem of queries (especially SPJ queries where SPJ stands for Selection-Projection-Join) is solvable.

(c) Nonprocedural expression is possible (easy to use).

(d) Result of a query is a relation which can be manipulated by other queries (closure property).

For complicated applications, use of objects becomes important. Data, methods and correspondences (between data and methods) should be stored separately, if conventional database systems are used to realize objectbases. Object-oriented database systems use IS-A structure for both data sharing and method sharing through inheritance. In this sense, data sharing and application property (method) are strongly combined, and thus efficient systems can be realized. Major objectives on object-oriented database research are adding flexibility such as supporting views.

Future trend is development of objectbase systems which are flexible. Currently the following three approaches are known.

(1) Realization of objectbase systems based on extended relational model (third generalization database approach [19][2], postgres [3], SQL3 [32])

(2) Extension of object-oriented databases to include relational databases as their subsets (uniSQL approach [31][34], ODMG-93 [33])

(3) Development of open database systems, so that object-oriented databases and relational databases (RDBs) can be combined together. Development of object-oriented interface for relational databases and relational interface for object-oriented databases are special cases.

[Requirement II] Development of data models to realize flexible objectbases.

Reduction of redundancy is important to reduce the maintenance cost (update, semantic constraint enforcement etc.). Mechanisms used for such a purpose are summarized in Table 3.

Data structure mainly corresponds to relation design. In object-oriented databases, links are combined with object identifiers. Inheritance is a convenient mechanism for data sharing. The IS-A relationship between subclass and superclass shows that attributes
and methods defined by the superclass are inherited by the subclass. Such an inheritance mechanism is class-based, which can realize data sharing at schema level.

![Figure 1: Data Hierarchy for Security Mechanism](image)

Since inheritance is very useful to reduce redundancy of both data and methods, it is suitable for object-oriented databases. Although most redundancy of data can be removed by those mechanisms, there remains redundancy of methods.

For example, a simple data hierarchy to realize a security mechanism is shown in Figure 1. Each method is considered to be a combination of a user-name and an operation. By Bell-Lapadula model[7], a person who can read data in one class can read data in lower classes. If user Tanaka can read confidential data, he can also read secret data. Thus, it can be expressed as inheritance of classes shown in Figure 1, since data in a lower class inherits all methods from a higher class. It is not easy to use object-oriented data model to realize such inheritance mechanisms. Since it is completely unrelated to IS-A hierarchy, we need many classes (direct product of IS-A hierarchy and security hierarchy) and thus it is impossible to use object-oriented databases[16]. Furthermore, for write operations, inheritance from a lower class to a higher class must be realized, since if Tanaka can write confidential data he can also write secret data.

Some authors discuss inheritance from a lower class to a higher class in IS-A hierarchy. It is not sufficient to reduce redundancy caused by methods. We need to develop mechanisms to reduce such redundancy.

[Requirement III ] Development of new inheritance mechanisms in order to reduce redundancy of methods.

Our objective is to develop a new data model which satisfies the above three requirements. Especially, we put emphasis on developing flexible and powerful objectbases.

3 Towards Flexible and Powerful Objectbases

In the previous section, we discussed three requirements for new data models. In this section, some specific problems of object-oriented data model are discussed in order to realize flexible and powerful objectbases.

1) Object Views

The concept of views was introduced to define customized interfaces for different applications. Views in relational databases can be defined easily using relational algebra. In order to achieve flexibility of object-oriented databases, the research on view mechanisms for OODB has been popular. However, most of the view mechanisms in OODB [30] [15] [14] [1] [25] [24] are realized by the almost same approaches as those used in relational databases: computation views and materialization views. It is difficult to realize autonomous views (objects in views having their own attributes and methods) suitable for objectbases, since they were implemented based on query processing. These methods do not provide the persistent identifiers for object views. The former is object preserving: the objects contained in the result of a query are the instances of the source classes. Thus, object views are actually the instances of the source classes. The latter provides object views with identifiers but the identifiers are temporarily generated and therefore are not persistent. A query are the instances of the source classes. As expressed in [8], providing persistent identifiers for object views is crucial if views are allowed to be defined additional attributes. However, the method proposed
in [8] faces view update problem. That is, the known view mechanisms of OODBs can achieve some flexibility but lose OODBs’ efficiency. In order to keep the efficiency of OODBs, object views with autonomous properties should be realized.

(2) View Updates

View update in RDBs is considered to be very difficult and only updates which contain all key attribute values in the view can be applied. It is mainly because tuples in RDBs have not unique identifiers and they can be divided and combined freely by relational operations. Especially, a project operation may merge several tuples. In OODBs, data are encapsulated in terms of objects and objects are identified independently of their status. Since two objects with the same states are still regarded to be different due to their different identifiers, the project operation cannot cause any information loss. Therefore, view update problem in OODBs is not so difficult as in RDBs. The view update problem in OODBs is usually solved through object-preserving [25]: the objects contained in the result of a query are the input objects. Object preservation makes updates propagate automatically: an update to an object in a view is an update of a base object. This way is only suitable for the views based on query. However, the view update problem is still open for the materialized views of OODBs [8].

(3) Object Roles

The concept of roles was introduced to represent the many faceted nature of real world entities. It is different from the concept of views. Roles usually require some additional attributes and methods. For example, consider a person who is both a student and an employee. As a student, a person needs an attribute for his grade, while as an employee, he should have an attribute for his salary. An extension of the network model used in the CODASYL database system has introduced the role-segment concept [6]. That is, the record represents the existence of a real-world entity while the role-segment represents the existence of one of the entity’s roles. Many OODBs restrict that each object is a direct instance of only one class and indirectly belongs to all superclass of its class. That is to say, an object cannot reside simultaneously in two classes between which are not related by a sequence of IS-A relationships. For example, a person cannot be defined as a student and a teacher at the same time. In order to overcome such a shortcoming, the multiple inheritance mechanism was introduced to define intersection class such as class StudentTeacher which contains objects corresponding to persons who are both students and teachers. But this approach may lead to a combinatorial explosion of sparsely populated intersection classes. Several object data models with roles [23][4] have been proposed to address this problem. These models allow an object to belong to several conceptual categories namely classes. An object belonging to a class is viewed as playing a role of that class. The set of roles possesses by an object can be changed by extension (role adding) and contraction (role dropping) operators. However, it is difficult to realize time-varying roles by these object role models.

(4) Object Migration

In order to capture the dynamic nature of real-world entities, the concept of object migration has been introduced. The obvious example of this kind of entity is a person. Throughout his or her life, a person may become student. After graduation, that person ceases to be a student, and becomes an employee. This problem is strongly related to above role multiplicity. There are some theoretical studies [13] on modeling the fact that not every sequence of object migration is admissible, it should be possible to specify admissible histories or migration paths. These theoretical results seem to be difficult to be implemented by the existing object role models, which only provide some special operators (extend/drop) to support simple object migration (role acquisition or role dropping).

(5) Multiple Inheritance

In order to allow a class to inherit attributes and methods from two or more source classes, the concept of multiple inheritance was introduced. It is not suitable for role multiplicity as discussed in (3) but useful for aggregation of roles of an object, such as data integration. However, as some of the attributes and methods is inherited, ambiguity can arise when more than one source class defines attributes and methods with the same name. The two most common ways of resolving ambiguity in multiple inheritance are graph-oriented and linear [28]. In the graph oriented approach, the system deals with the IS-A relationship graph directly. Systems using this approach often given an error if a conflict occurs in inheritance. An alternative linear approach flattens out the IS-A relationship graph, using some ordering on the superclasses of a subclass. In addition, there are some systems forbidding the creation of a hierarchy in which there is ambiguity or forcing explicit resolution through the definition of an extra method. That is, there are many possible solutions for dealing with the problem of conflict resolution.

(6) Data-Knowledge Coordination

As have been described, data and knowledge are strongly combined in OODBs, where knowledge are expressed as methods. This way can achieve efficiency but flexibility will be lost. Data-knowledge coordination model is introduced to combine various
kinds of databases and knowledge bases under distributed environment where autonomy of each system is important[11]. It emphasizes separation of data and knowledge and their dynamic coupling. As shown in Figure 2, it has three hierarchies, one for data, one for knowledge and one for relationships between data and methods. Great flexibility can be achieved by these three kinds of hierarchies. It is suitable for to develop applications over independent databases and knowledge bases. It can be also used as a model to realize flexible objectbases. However, if coordination among data and knowledge are complicated, the model requires high computational overhead, that is, a system based on the model may not be efficient.

4 Object Deputy Model

4.1 Deputy Objects and Deputy Classes

Object deputy model shares the encapsulation feature of object-oriented data model. It models real-world entities in term of objects. Objects are identified by system-defined identifiers which are independent of their states. An object has some attributes which represent properties of its corresponding real-world entity. The state of an object is represented by its attribute values, which are operated by methods.

[Definition 1] An object is defined as

\[ o = (o, \{T_a : a\}, \{m : \{T_p : p\}\}) \]

1. Identifier of object \( o \) is directly represented by the symbol \( o \).
2. \( T_a : a \) is used to represent an attribute of object \( o \), where \( a \) and \( T_a \) are attribute name and type, respectively. The value of attribute \( a \) of object \( o \) is expressed by \( o. a \). For each attribute \( T_a : a \), there are two basic methods, defined as follows.

\[ \text{read}(o, a) \Rightarrow \uparrow v.a, \]
\[ \text{write}(o, a, v) \Rightarrow o.a := v \]

where \( \Rightarrow \) stands for operation invoking, \( \uparrow \) stands for returning result value and \( := \) stands for assignment.
3. \( m : \{T_p : p\} \) is used to represent a method of object \( o \), where \( m \) and \( \{T_p : p\} \) are method name and a set of parameters. \( p \) and \( T_p \) represent parameter name and type, respectively. Applying \( m \) is expressed as follows.

\[ \text{apply}(o, m, \{p\}) \]

Classifying objects having the same attributes and methods into classes in databases helps avoid specification and storage of much redundant information. Like object-oriented data model, object deputy model clusters "similar" objects together into a class. All objects belonging to the same class are described by the same attributes and the same methods.

[Definition 2] A class is defined as

\[ C = \{\{o_i\}, \{T_j : a_j\}, \{m_k : \{T_k : p_k\}\}\} \]

1. \( \{o_i\} \) : the set of objects, the extension of \( C \).
2. \( \{T_j : a_j\} \) : the set of attributes of objects in \( C \).
3. \( \{m_k : \{T_k : p_k\}\} \) : the set of methods in \( C \).
The same real-world entity can be classified in multiple ways. For example, a student can be classified into graduates and undergraduates. He or she can also be classified into foreign students and domestic students. This example shows that each real-world entity can belong to multiple classes. However, most of object-oriented data models restrict that an object belongs to only one class. Object deputy model introduces the concept of deputy objects and allows an object to have multiple deputy objects which belong to different classes.

[Definition 3] Let $o$ be an object. One of its deputy objects, named as $o^d$, is defined as follows.

1. Deputy object $o^d$ has its own identifier, attributes and methods.

2. There is a bilateral link between $o^d$ and $o$, which is represented by the mapping $f$ and its inversion $f^{-1}$. $f$ and $f^{-1}$ are defined as

$$f(o) = o^d, \quad f^{-1}(o^d) = o$$

3. Attribute $T_j : a_j$ of $o$ can be used as if it would be attribute $T_j^d : a_j^d$ of deputy object $o^d$ if the following switching operations are defined for $o^d$.

$$\text{read}(o^d, a_j^d) \Rightarrow f_{T_j \mapsto T_j^d}(\text{read}(o, a_j))$$
$$\text{write}(o^d, a_j^d, v_j^d) = \text{write}(o, a_j, f_{T_j \mapsto T_j^d}(v_j^d))$$

As shown in Figure 3, read or write operation on $a_j^d$ of deputy object $o^d$ is really applied on the corresponding attribute $a_j$ of object $o$. In addition, the names and types of the attribute between object $o$ and its deputy object $o^d$ are allowed to be different by switching operations and conversion functions($f_{T_j \mapsto T_j^d}, f_{T_j^d \mapsto T_j}$).
4. Method \( m_k : \{T_k : p_k\} \) of \( o \) can be used as if it would be method \( m^d_k : \{T^d_k : p^d_k\} \) of deputy object \( o^d \) if the following switching method is defined for \( o^d \):

\[
\text{apply}(o^d, m^d_k, \{p^d_k\}) \Rightarrow \text{apply}(o, m_k, \{f_{T^d_k \leftarrow T_k}(p^d_k)\})
\]

As shown in Figure 4, method \( m^d_k \) applied on deputy object \( o^d \) is really transformed into applying method \( m_k \) of object \( o \). The names of the method and types of its parameters between object \( o \) and its deputy object \( o^d \) are allowed to be different by switching operation and conversion functions (\( \{f_{T^d_k \leftarrow T_k}\} \)).

An object can have multiple deputy objects. On the other hand, multiple objects can share a single deputy object. A deputy object of more than one object belonging to different classes can be used for aggregation abstraction. An aggregation abstraction defines a complex object from more than one other object that represents one of its component parts. For example, a bicycle can be defined from wheels, pedal, handlebar, and so forth.

Furthermore, a deputy object can be used for grouping instances of a class satisfying some condition. Grouping objects by deputy objects is different from clustering objects by classes. Consider such an example that the persons of the same age are grouped into a group. Since the number of ages is not fixing, the number of the groups is changeable. Obviously, we need not give a name for each group. However, each class must be assigned with a unique class name. The class cannot be dynamically created and deleted according to state of database. A deputy object used for grouping has a set of source objects satisfying some grouping condition and it can appear or disappear over the time. There is another difference. The class is used to classify objects having common attributes and methods while the deputy object is used to group objects having the same values of some attributes. That is to say, the class is for grouping objects according to their schemata while the deputy object is according to their states.

The schemata of deputy objects are defined by deputy classes.

[Definition 4] Let \( C^o = \{o^o_i\}, \{T^o_i : a^o_i\}, \{m^o_i : \{T^o_k : p^o_k\}\} \) be a class and \( C^d \) a deputy class of \( C^o \). \( C^d \) is then called one of source classes of \( C^d \). Deputy class \( C^d \) is defined as

\[
C^d = \{o^d_i | o^d_i \to o^o_i \times \ldots \times o^o_i \times \ldots | \{a^o_i\},
sp(o^d_i) | cp(\ldots \times a^o_i \times \ldots) | \text{gp}(\{a^o_i\}) == \text{true}\}
\]

1. \( \{o^d_i\} \): the set of deputy objects, called the extension of \( C^d \), where \( o^d_i \to o^o_i \times \ldots \times o^o_i \times \ldots | \{a^o_i\} \) represents that \( o^d_i \) is the deputy object of \( o^o_i \times \ldots \times o^o_i \times \ldots \)

or \( \{a^o_i\} \). \( sp \), \( cp \) and \( gp \) represent selection, combination and grouping predicate, respectively.

2. \( \{T^d_i : a^d_i\} \cup \{T^d_j : a^d_j\} \): the set of attributes of \( C^d \), where \( \{T^d_i : a^d_i\} \) is the set of the inherited attributes and \( \{T^j_i : a^d_j\} \) is the set of the additional attributes of \( C^d \).

3. \( \{m^d_i : \{T^d_k : p^d_k\}\} \cup \{m^d_j : \{T^d_k : p^d_k\}\} \): the set of methods of \( C^d \), where \( \{m^d_i : \{T^d_k : p^d_k\}\} \) is the set of the inherited methods and \( \{m^d_j : \{T^d_k : p^d_k\}\} \) is the set of the additional methods of \( C^d \).

4.2 Object Deputy Algebra

Object deputy model provides an object deputy algebra for proxy class derivation, which consists of six operations: Select, Project, Extend, Union, Join and Grouping. They have closure property and can be used for data abstraction by deriving deputy classes as below.

Specialization is an abstraction process to define a subclass of a class called superclass. The instances of the subclass is a subset of the instances of the superclass. The subclass shares attributes and methods of the superclass and also has additional attributes and methods. The subclass can be defined by the deputy class derived by algebraic operation Select. The Select operation is used to select instances from the source class according to selection predicate and create their deputy objects as instances of the derived deputy class. The additional attributes/methods are associated with the deputy objects of the selected objects by the algebraic operation Extend.

Generalization is a reverse process of specialization which suppresses the differences among several classes, abstracts their common attributes/methods and generalizes them into a single superclass. The superclass can be defined by the deputy class derived by the algebraic operation Union. The Union operation creates deputy objects of instances of several classes as instances of the derived deputy class and restricts the derived deputy class to only inherit the common attributes/methods from the source classes.

Aggregation is an abstraction process that defines a complex class from a set of other classes that represent its component parts. The complex class can be defined by the deputy class derived by the algebraic
operations Join. The Join operation assembles related component classes into a complex class according to combination predicate.

Grouping is an abstract process that groups instances of a class into groups, which are defined as instances of another class, called grouping class. The grouping class can be defined by the deputy class derived by the algebraic operation Grouping. The Grouping operation groups instances of source class according to grouping predicate and creates deputy objects of the groups as instances of the derived deputy class.

In addition, a SQL-like definition language is designed based on the object deputy algebra. It provides five statements for defining base class for classification, and deputy classes for specialization, generalization, aggregation and grouping of real-world entities, respectively.

4.3 Constraints on Objects and Deputy Objects in the Definition

There are dependency constraints between objects and their deputy objects that are defined as predicates in deputy classes. Only when an object satisfies the predicate, its deputy object can exist. If the object is deleted or its state is updated so that it cannot satisfy the predicate, the deputy object may be deleted. In addition, the existence of a deputy object may depend on its environment. For example, whether the deputy object of a geographic object appears in the displayed map depends on the size of the map, number of deputy objects and priority ranks. Because deputy object has its own deputy object, the latter may exist although the former does not satisfy its existence condition. In this situation, the former cannot be deleted since the latter may inherit some attributes and methods from the former. In order to solve this problem, we allow each deputy object to have a flag that represents its active/inactive state and is controlled by methods (application property). Furthermore, in order to avoid generating deputy objects that have not any relationship with their source objects, we require that some dependency constraints on key attributes between objects and their deputy objects should be defined. For example, names of a person and its deputy objects as a student and a son should be kept consistent.

In order to enforce integrity constraints, we have designed three procedures for data update propagation, which caused by basic update operations: addition of an object, deletion of an object and modification of attribute value of an object, respectively. They are realized based on dependency constraints defined by deputy classes. Through data update propagation, dynamic classification can be supported. Object deputy model allows methods to be defined to watch states of data in databases. That means that methods can be triggered when their watching data are updated. Data update propagation may cause method application propagation that can be used to realize active database functions.

5 Modeling Capabilities of Object Deputy Model

5.1 Generalized Inheritance Mechanism

As shown in Figure 5, two conventional inheritance mechanisms: IS-A and Delegation can be generalized in object deputy model.

In object-oriented databases, the inheritance mechanism is realized based on IS-A relationship between subclass and superclass. When a message is sent to an instance of the subclass, the method for executing the message is searched in the subclass. If the method is not found, the direct superclass of the subclass is navigated into through the pointer from the subclass to its superclass and the method searching is continued in the superclass until the method is found. Such an inheritance mechanism is class-based, which can realize data sharing at schema level.

In object-oriented databases, the inheritance mechanism is realized based on IS-A relationship between subclass and superclass. When a message is sent to an instance of the subclass, the method for executing the message is searched in the subclass. If the method is not found, the direct superclass of the subclass is navigated into through the pointer from the subclass to its superclass and the method searching is continued in the superclass until the method is found. Such an inheritance mechanism is class-based, which can realize data sharing at schema level.

There is another inheritance mechanism called delegation[18][27], which allows data sharing at instance level. It does not define classes. An object is not an instance of some class, but rather it is simple a collection of named slots. A slot may contain another object or a block of codes. When any message is sent to an object, the receiver determines if it has a slot corresponding to the message. If the slot contains a value, then it returns that value. If the slot contains a block of codes, the block is executed and the result returned. If the object does not have a slot correspond-
ing to the message, it delegates the message to some other object. Typically, the message is forwarded to the object contained in a designated parent slot of the receiver.

In object deputy model, inheritance is realized by switching operations that can forward the messages received by the deputy object to its source object through the pointer from the former to the latter. The IS-A relationship between subclass and superclass can be completely simulated by defining such a switching operation: when a deputy object receives a message, if the additional methods of the deputy object cannot execute the message, the message is forwarded to the source object through the pointer from the deputy object to the source object. The method for executing the message will be searched in the source class. On the other hand, the inheritance between deputy object and source object is actually a kind of delegation, which is controlled by switching operations. The switching operations defined by deputy classes can model various semantics of delegation. Therefore, the inheritance mechanism provided by object deputy model is generalization of IS-A relationship and delegation. It can achieve data sharing at both schema and instance levels.

Not all of attributes/methods of the source object can be inherited by its deputy objects. The inheritance is defined by application property that is expressed by switching operation. Inheritance by switching operations can provide stronger hiding capability than the IS-A relationship, which is useful for data security. By the latter, if we want to hide an attribute of the superclass, we have to define an additional attribute with the same name in the subclass, which can override the value of the attribute defined by the superclass. But actually, the attribute name can still be seen through the subclass. However, by the former, we can hide both attribute name and attribute value by not defining switching operation for it.

In addition, switching operations allow inheritance to be controlled by application environment. Suppose geographic objects are displayed in maps through their deputy objects. Deputy objects should inherit sizes of geographic objects according to number of the displayed objects and size of the map. The ratio of the displayed size to the actual one can be dynamically computed by switching operations, which retrieve the actual sizes from geographic objects and multiply them by the radio so that the proper sizes of geographic objects can be used by their deputy objects.

Besides the inherited attributes/methods, deputy objects have the additional ones. If both of some of them represent the same property, they can be used selectively by switching operations. For example, a traveling agent sells tickets for an airline company. It has tickets from the company and can ask more if its holding tickets are sold out. That is, it has its own available ticket attribute and can inherit the available ticket attribute from the company. If the value of the former becomes zero, the agent will use the value of the latter. This can be defined by switch operation that defines such a semantic constraint on inheritance: the agent's own attribute is first read; if the value is zero, the attribute of the company will be read. Sometimes, the inherited attribute is used prior to the additional one. For example, an employee has his own work and also has to do some work as deputy of his boss. If there is time conflict, he has to do his boss's work firstly. This can also be defined by switching operation with such a semantic constraint on inheritance: whether there is any work from his boss is seen; if there is not, his own work will be done.

5.2 Unified Realization of Object Views, Roles, Migration and Multiple Inheritance

Object deputy model can provide a unified realization of object views, roles and migration[21]. As described in Section 3, the previous researches on these concepts were carried out separately. In fact, they are very closely related. Objects appearing in a view can be regarded as playing roles in that view. Object migration is caused by change of roles of an object. Multiple inheritance realizes aggregation of object roles.

The unified treatment of object views, roles, migration and multiple inheritance can achieve the following advantages.

(1) Treating object views as roles of an object allows object views to have additional attributes and methods so that autonomous views required by advanced applications can be realized.

(2) Handling object roles in the same way as object views enables object migration to be easily realized by dynamic classification functions of object views.

(3) Generalization and unification of object views, roles, migration and multiple inheritance make it possible that various semantic constraints on them can be defined and enforced uniformly.

Deputy objects can be used for such a purpose. Since deputy objects can be used as customized interfaces of source objects by selectively inheriting the attributes and methods of source objects, they can be used to realize object views. Furthermore, the views defined by deputy objects are updatable. Since a deputy object and its source object are linked bilaterally, the update on the deputy object can be propagated to the source one and the update on the source
object can be reflected to its deputy object. On the other hand, deputy objects can be extended with additional attributes and methods, and therefore they can play roles instead of their source objects. By means of dynamic classification functions of object views supported by update propagations, handling object roles in the same way as object views can realize time-varying roles, which are actually regarded as object migration. Not all of object migrations are admissible. Restriction on object migration are defined by various predicates and dynamically maintained through data update propagation. For example, so long as an object satisfies the selection predicate of a deputy class, it will be migrated into the deputy class by creating an instance in it. Such a migration is restricted by the selection predicate of the deputy class.

Supporting object migration requires to record the history of an object. By object deputy model, a role of an object is played through one of its deputy objects. Both objects and their deputy objects are defined to have two system-defined attributes. One is named as tf and used to record the time at which the object(deputy object) is created. The other is named as t and is used to record the time at which the object(deputy object) is deleted. The value of attribute t of an object(deputy object) is kept to be ∞ until it is deleted.

Defining time dimensions of a source object and deputy object separately allows to represent some sequences of object migration which are difficult to be represented by conventional approaches. For example, we can represent the following case: An object has been migrated into another class but the object will stay in the original class for a while. In this situation, the two deputy objects (o1, o2) of the object belonging to the original class and target class respectively have the relationship: o1 > o2 in the same way allows the deleted objects(deputy objects) to be remained in databases. It can avoid dangling references. Furthermore, we can express such a situation that an object leaves from some class and may enter the class once again afterwards. For example, some employee may be sent to work in another company for one year and then return to its original position. When the employee leaves, the o1 of its deputy object in the original position is assigned with that time. Once the employee returns, its deleted deputy object in the original position can be restored to be active by changing o1 into ∞.

Object deputy model can also support multiple inheritance, since a deputy object is allowed to have more than one source object belonging to different classes. It resolves ambiguity in multiple inheritance by forcing explicit resolution through switching operations. For example, a person works for university and company, which are respectively defined as classes A and B. Suppose a class C is defined as a deputy class of A and B to represent intersection of teacher and employee. Both A and B have defined a salary attribute. Defining proper switching operations allows A’s and B’s salary attributes to be inherited by C in one of the following ways:

1. Both A’s and B’s are inherited by integrating them using computation function, such as the function computing their average value.
2. One of A’s and B’s is inherited and the other is ignored. If the person mainly works for company, the salary attribute of C can be selectively defined to be inherited only from B’s while the inheritance of A’s is to be forbidden.
3. A’s or B’s is selected to be inherited according to their values. Defining the salary attribute of C as the maximum of A’s and B’s can be realized by selecting the maximal one through switching operations.

6 Application Examples

6.1 Flexible Multibase System

A flexible multibase system can be realized by combining object deputy model and data-knowledge coordination model. Data-knowledge coordination model is suitable for integration of databases and knowledge bases under distributed environment. Since data and knowledge are classified in different ways: data by their attributes and knowledge by their subjects, the model introduces the concept of context modules to realize dynamic coupling of the independently managed data and knowledge. It consists of three kinds of hierarchies: data object hierarchies for data integration, knowledge object hierarchies for knowledge integration, and context module hierarchies for data-knowledge coordination. Through these three kinds of hierarchies, data and knowledge can be properly tailored and dynamically coupled to fit different applications. The data-knowledge coordination model has the following features.

Autonomy: The autonomy of the underlying DBs and KBs needs be retained as much as possible. On the other hand, the imported data and knowledge in context modules should have enough autonomy required by realization of efficient applications.

Multiple Viewpoints: Data and knowledge can be tailored for different applications.

Dynamic Classification: Coupling of data and knowledge can dynamically be determined according to semantic constraints on the context modules.
We have implemented this model using an object deputy system in Smalltalk. The organization is shown in Figure 6 [22]. Interpretive nature of Smalltalk can realize dynamic links among multiple databases (DBs) and knowledge bases (KBs). Through these links, the independently managed data and knowledge are imported into and treated as objects in Smalltalk, where data and knowledge are integrated by their respective deputy object hierarchies which can be constructed by object deputy algebra. The context module hierarchies for coordination of the integrated data and knowledge are realized based on the model-view-controller architecture of Smalltalk.

In this implementation, context modules consist of deputy objects of the underlying data and knowledge. Since the deputy objects are allowed to have additional attributes and methods, they can have enough autonomy so that applications can be realized efficiently. When the same data/knowledge object is used in more than one context module, it can be defined to have more than one deputy object, one for each context module. In addition, a deputy object of the underlying data/knowledge object can be tailored by switch operations and therefore multiple viewpoints of the underlying data/knowledge object can be realized through their multiple deputy objects. Since update propagation can be realized based on the bilateral links between the deputy objects and their source objects, realization of dynamic classification is also possible.

6.2 Geographic Database

Object deputy model is useful in geographic databases [5]. We assume that geographic objects are stored in geographic databases. A map is realized by displaying deputy objects of the geographic objects as shown in Figure 8.

Due to the limited resolution of a display, the number of objects to be displayed is limited. Consequently, priority is required to be assigned to each object, and objects to be displayed are determined by the priority. In order to displaying a same geographic object with different priorities in different maps, deputy objects are used and the different priorities are separately stored in the additional attributes of its different deputy objects. Besides the additional attributes, some attributes in the deputy objects are inherited from geographic objects by switching operations. If the shape of the geographic object in the map is determined by the property of the geographic object, the value of attribute shape of a deputy object is determined by applying switching operation to the property value of the source object. Locations in the displayed map are examples of attribute values only for deputy objects.

From the displayed map, its sub-maps can also be
derived by defining deputy objects of deputy objects of geographic objects as shown in Figure 7. Whether a deputy object appears in the displayed map depends on its priority, the size of the map and number of deputy objects. A flag bit is added to each deputy object, which shows active/inactive of the object. If it is inactive, it will not be shown in a map while it may be shown in a submap.

6.3 VIEW Office

In order to support cooperative work based on database systems, an office system called VIEW Office is under development as shown in Figure 9 (previously it was called VirtualOffice[29]).

In an office, there can be more than one project. We can define office spaces for each project (shared spaces) and for each user (private spaces). A user can define several spaces, each for each project. Users in one project can work together to accomplish the project. Cooperation of users is realized through shared data and knowledge. If a data/knowledge object is involved in two projects, we can define a deputy object for each project. These deputy objects can be used independently in their respective projects. Since there is authorization mechanism for deputy objects and the important update operation can be mediated through the source object, data security in VIEW Office can be supported.
English, he cannot read research reports in Japanese. Deputy objects of a research report can change its presentation forms by switching operations. Research reports can be authorized to be updated in office spaces. The update on the personalized parts of a research report is only restricted in the user's private space. The update can propagated among office spaces if the updated parts are shared by users working in these office spaces.

Since the users themselves are regarded as objects, we can use object deputy model to define deputies of users. When a user participates in multiple projects, the work can be made more effective if one deputy object of the user is defined for each project. Each deputy object corresponding to a role can be regarded as a virtual person.

6.4 VIEW Classroom

Deputy objects can also be used for presentation in VIEW Classroom - a distant education system[17] as shown in Figure 10.

![Diagram](image)

Figure 10: Presentation in VIEW Classroom

We assume presentation materials are stored as objects in objectbases. They are presented in displays of teacher and students in different forms. A teacher may require additional teaching materials which are not shown to students. For students, questions-answers of the same class in previous years may be added. This can be realized by different deputy objects for a teacher and students of the same presentation object. The deputy objects of the deputy object for students can be defined so that the presentation can satisfy different requirements of students, for example, having different physical handicaps. In addition, students can generate his own notes by personalizing the teaching materials.

7 Concluding Remarks

As shown in examples shown in Section 7, we may have duplication of data and methods in many applications. Inheritance based on IS-A is not sufficient to reduce such redundancy. Deputy objects offer mechanisms to share data and methods in order to reduce redundancy. As inheritance mechanisms offered by deputy objects can be defined independently from IS-A hierarchy, flexible objectbases can be realized by the deputy object model. In order to keep advantages of object-oriented databases, inheritance by deputy objects should be used if inheritance by IS-A hierarchy cannot be realized. We believe that the object deputy model is suitable to realize flexible objectbases.

References


[34] Won Kim, UniSQL/X Unified Relational and Object-Oriented Database System, SIGMOD'94 (1994).