

A View Class Mechanism for Object-Oriented Database Systems

Yutaka Kimura

Kunitoshi Tsuruoka

C&C Systems Research Laboratories
NEC Corporation

4-1-1 Miyazaki, Miyamae-ku, Kawasaki, Kanagawa 216, Japan

Abstract

Object-oriented database systems generally tend to impose a single, structural view on users. This paper proposes a new mechanism, called *view class*, which realizes various views for object-oriented databases. The view class, created by query primitives and additional methods, enables information hiding as well as the ability to provide a join view or a group view. In addition to the language for view class definition and manipulation, an implementation scheme is given. The scheme makes it possible to hide methods (called method projection), or to determine a method uniquely where an instance object can belong to several classes. A view anomaly problem, which occurs due to introduction of the method projection and its solution, are presented. Also shown are some problems which occur when a database is updated through a view. They are solved by making direct reference to source objects through view classes, and by storing update semantics in the form of methods.

1 Introduction

Object-oriented database systems are becoming important in some fields, such as CAD, CASE, and office automation with multi-media information. A lot of implementation efforts for object-oriented DBMS have been made. However, they lack some features or advantages of relational (conventional) database systems. For example, data independence by views, dynamic operations for making object relationship, such as join or group, are not found in them. The authors are developing an object-oriented database management system, called *Odin*, which includes the features described above in the framework of an object-oriented data model. This paper proposes an extended object-oriented data model, including the "view" concept and its implementation.

A few papers describe the view for object-oriented database systems [TaYI88, ShZd89, TaCh89]. [TaYI88] suggests a virtual class and a virtual schema. A virtual class is a procedure for returning objects which satisfy a condition from the underlying classes. A virtual schema is an IS-A hierarchy between the virtual classes. The difference between a real class and a virtual class is that it is possible for an instance to belong to several virtual classes. However, no methods can be hidden from a class. Moreover, join views or group views are not considered. [ShZd89] introduces relational operators, such as selection, join, and so on, into an object-oriented data model. These operations generate new objects, which are related with existing objects, and operations return a set object, whose elements are the new objects. Natural join operation is also considered in [TaCh89]. The result is a new complex object, into which two complex objects are joined recursively. However, these papers do not mention a new class generated by the operations, nor methods for new objects. In the *Odin* data model, new objects are created, which make direct reference to existing objects, and the operations return a set object. Furthermore, a new class, called a *view class*, is generated, whose instances are the elements of the set object.

The notion of view class, proposed in this paper, is somewhat similar to the virtual class in [TaYI88]. However, the mechanism has the following original features.

1. A view class is flexibly created through query primitives, such as select, join, and group, in an extension of C++.
2. It provides a view consisting of more than one class or more than one instance through join or group operations.
3. It can hide instance methods (or instance variables) using a projection function, called *method projection*.
4. It includes a view checking mechanism, because the method projection causes the occurrence of a new problem, called *view anomaly*, wherein the projected method may invoke other methods that are not projected.
5. Update problems through a view class are simply resolved, since view instances directly refer to source instances, and since update semantics can be specified in the view class as methods.

```

useclass Text;
permanent
class Module {
:public
    string  name;
    int     version;
    Text    source;
}
int Module::no_of_lines(){ /* body */ }
permanent
class Bug_Report {
:public
    int     product_version;
    string  bug_module;
    string  cause;
}

```

Figure 1: Example Database

2 View Classes

2.1 Basic Concepts

A basic object-oriented data model is based on [ABDD.89]. However, a set object and its operations are shown here, since the paper does not describe a set operation in detail.

A *set object* is defined which contains a set of objects in an instance variable, and it is instantiated through a set class. A class has a set object to manage its own instances. Operation of a set of objects is implemented to send a message to the set object. The query result is a new set object, whose elements satisfy the specified conditions. The methods for a set object are select, join, and group. *Select* returns a set object, which holds objects satisfying the selection condition. *Join* is used to view some source classes as a single (joined) class. It generates tuple objects to relate objects, which satisfy the join condition, and returns a set object, which holds the tuples. Given a set object of the target class for grouping, and instance variables which designate grouping unit, *group* generates set objects, which hold objects having the same value as the instance variable specified. Afterwards, it returns a set object which holds grouped set objects.

A view concept is introduced into the basic model by some extension. In the remainder of this paper, a simple database is assumed. The database has two classes, Module and Bug_Report, as shown in Fig. 1. The keyword "permanent" means that instances generated from the class may be persistent, and the keyword "useclass" indicates that this program will reference a specified class, which has already been defined in the database. A class Module instance represents a function for a source program, and a class Bug_Report instance is a report, which is written when a bug occurs.

One characteristic is that a view class can restrict reference to a set of instances in a class, instance variables, and methods. Through the *selection view* generated by a select operation, users can only see a subset of instances in a source class, and the variables and methods which are specified in its visible condition. Figure 2 is an example of selection view definition. This figure indicates that view class "Module_for_Debug" has instances having the module name prefix

"Debug.". A method for instantiating view instances is described in a derive method. First, a set object of class "Module" is obtained by \$ operator, and then a select message is sent to it with a parameter. A syntax for the parameter is extended as a query, which can be declaratively written in C++. The parameter enclosed by brackets is called a *block parameter*. A block parameter consists of object variable declaration, view-clause, and where-clause (somewhat like SQL syntax). In object variable declaration, an object variable corresponding to a class is declared. The object variable holds an object to which a block parameter is applied. The "self" in the declaration represents a receiver object, itself. The where-clause is an expression representing a selection condition. A set of instances, obtained by a select operation, can only access through instance variables and methods in view-clause. The selection condition for the example means that the query result is a set of instances having the module name prefix "Debug.". However, the instance variables and methods, which can be accessed through the view, are only "name", and "no_of_lines". The others, such as version and source, are not visible. Such a function, called *method projection*, is specified with a view-clause. On the other hand, in a relational data model, the attributes for a relation are restricted by projection operation. The projection copies the values of the specified attributes into a new relation with duplicate elimination. It is different from Odin data model, which does not generate any instances, but instead checks the access authorization at execution time.

Another characteristic is that the view class can regard one or more classes as only one class. Assume that users would like to analyze bug reports by modules. In this case, they should know the specifications for two classes, and should make queries to each class. If a view "Bug Reports by Module"(BRM) is supported, they need to know and manipulate only one class. That is, a view class has to be manipulated as if it were a single conceptual class. The BRM user may manipulate all the instance variables and methods for class Module and Bug_Report. In the above example, a view BRM is defined as shown in Fig. 3. The expression in the body of the derive method joins two classes, Module and Bug_Report, under a join condition in the where-clause. Users can virtually see one class, called a *join view*. It is the same as the relational data model to some extent, but implementation and update semantics are different, as described

```

view
class Module_for_Debug {
:public
    meta void derive();
}
meta void
Module_for_Debug::derive()
{
    $Module.select( [self: S; |
                    view S(name; no_of_lines);
                    where S.name == 'Debug_*';] );
}

```

Figure 2: Visibility Condition Example

in the following section.

In the relational model, a view is a query evaluated, when it is referred to by user's requests. In the proposed object-oriented model, a view is a method named "derive". A user defines a class name and the derive method. When he manipulates a database through the view class, he first sends a derive message and then a message for query.

```
view
class BRM {
:public
    meta void derive();
}
meta void
BRM::derive()
{
    $Module.join( [self: S; Bug_Report: B; |
        where S.name == B.bug_module] );
}
```

Figure 3: A View Class Example

2.2 Basic Model Extension

In order to realize the functions specified above, the basic object-oriented data model is extended. Definition 1 specifies relationships between classes and instances (1-4), and view methods (5-7).

Definition 1:

1. An object must belong to only one conceptual class.
2. An object may belong to several view classes.
3. No instance variable can be defined for a view class.
4. View classes are direct subclasses of a top class "Object".
5. Methods for view classes may be defined. They are called *additional methods*.
6. All the methods in the source classes are propagated into a view class. Such methods are called *primitive methods*.
7. Primitive methods available in a view class are those specified as visible in the derive method.

2.3 Operations

This section describes how three operations, select, join, and group, are implemented as view, and how a method is determined when a message is sent to an instance which belongs to a conceptual class and view classes. Examples in Fig. 2 and Fig. 3 are used in the following discussions.

Select

Implementation of a view class, "Module_for_Debug" in Fig. 2, is illustrated in Fig. 4. A view class primarily manages three kinds of information; visible methods, additional method definitions, and an <instance-set>. The visible methods have a list of method names (including access methods to instance variables), which are visible to the view users.

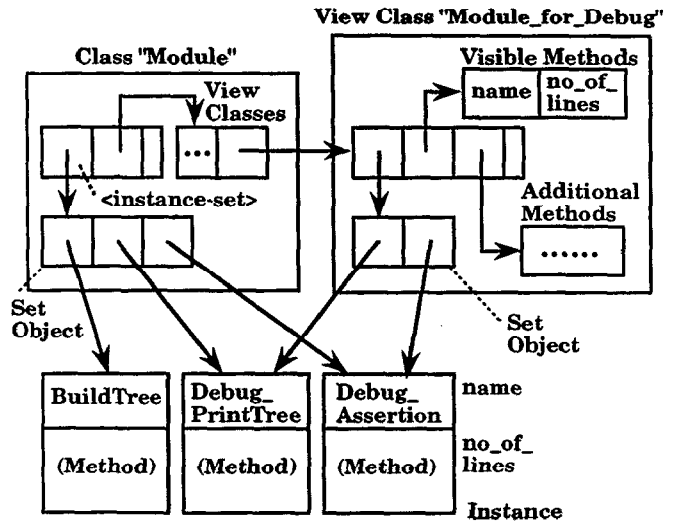


Figure 4: Select View Implementation

They are indicated by the view-clause for a select message. The additional methods are specific methods for the view class. <instance-set> is an instance variable, which points to the set object for the view class. A conceptual class points to view classes, which are created by sending select messages to the conceptual class. On the other hand, each instance has an inverse pointer only to the conceptual class to which it belongs. Unlike relational databases, the selection makes only a set object, whose elements are a subset of the source class instances, and visible condition (projection) does not influence the source class instances.

In this environment, assume that a transaction T1 sends a certain message to an instance. The algorithm shows how a corresponding method is selected (see also Fig. 4).

1. Find a conceptual class through the class id, which is a part of an instance object id.
2. Check whether there are view classes referred from T1 in the conceptual class.
3. If several view classes exist, select the most recently generated one, and check the additional methods defined in the view class, to determine whether or not the corresponding method exists.
4. If such a method is found in the additional methods, it is activated. Otherwise, check the visible methods for the view class. If it is not found in the visible methods, return an error.
5. If it exists in the visible methods, find the corresponding method in the conceptual class and activate it.

Join and Group

A view class BRM in Fig. 3 is considered here. A view class "BRM" has Module and Bug_Report pairs instances, as illustrated in Fig. 5. The view class is instantiated by sending a derive message to the BRM class. A join method produces new instances (called *tuple instances*), in order to combine class Module instances with those for class Bug_Report. As described in Section 2.1, methods used for both classes

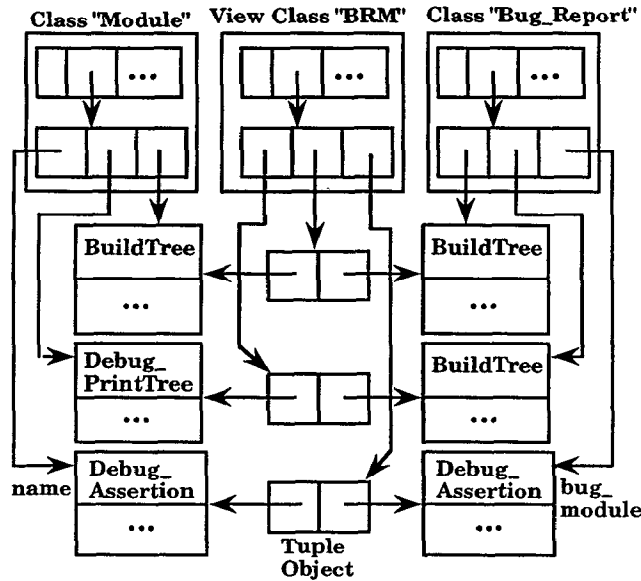


Figure 5: Join View Implementation

should be used for the tuple instances. To do so, the view class methods are automatically defined as having the same name as source classes methods, called primitive methods. Each only sends a message, which has the same name as itself, to a source instance. The source instance is the one pointed by an instance variable, whose domain is a source class, in a tuple instance. Notice that a tuple instance directly points to Module and Bug_Report instances, and that no data copy occurs from the contents of a Module or a Bug_Report instance.

In this case, assume that a transaction T1 sends a certain message to a BRM instance (tuple instance). The following shows how a corresponding method is selected.

1. Check whether or not there is a corresponding method in the view class, to which the tuple instance belongs.
2. If such a method exists, activate it. Otherwise, return an error.

In the example above, the selected method sends a message either to a Module instance or to a Bug_Report instance. As a result, a source class method is activated.

As a group message example, a view class "Bug_by_Cause" is considered, whose instances are generated by grouping "Bug_Report" instances by cause. This is defined as follows:

```
$Bug_Report.group( 'cause' );
```

Instances (set objects) are generated, one for every "cause" group, which points to Module instances having the same "cause".

The following Algorithm 1 shows a general method search procedure in Odin. Considered here is the method search for nested view classes as well as for conceptual classes. In the algorithm, a parent class may be a conceptual, join view, or group view class, and a transaction T1 sends a certain message to a target instance.

Algorithm 1:

1. Find a parent class, to which the target instance belongs.
2. Check whether or not there are selection view classes referred from T1 in the parent class. If none exists, go to step 5.
3. If several selection view classes exist, choose the most recently generated one, and check the additional methods defined in the view class, to determine whether or not the method corresponding to the message exists.
4. If such a method is found, activate it. Otherwise, check the visible methods for the view class, to determine whether or not the corresponding method exists. If it is found, activate it. Otherwise, return an error.
5. Check whether or not there is an additional method corresponding to the message (in this case, the parent class is a join view or a group view).
6. If the additional method exists, activate it. Otherwise, activate a primitive method.

A primitive method may send a message to another instance, which the target instance points to in Step 6. So, Algorithm 1 is applied to the new instance again.

3 View Anomaly

As mentioned earlier, the view method features are that view methods are propagated from the source classes for the view, and references for the methods can be restricted by select or join operations (called method projection). However, in order to check whether or not a view method is correctly defined, it is not sufficient to use a mechanism which ordinary object-oriented database systems or object-oriented programming languages include. In these systems, a check on a method uses correspondance of a method name, a class to which it belongs, parameter types, the number of parameters, and the type of return value. Assume that class V includes methods m1 and m2, and that class W includes methods m3 and m4. Furthermore, assume that m1 invokes m2 and m3. In such a case, if m1 and m2 are specified in a visible condition, when V and W are joined, m3 and m4 cannot reference through a join view. However, if m1 were indeed executed, m3 would be referenced in the m1. This is a contradiction with the visible condition specified. This contradiction is called *view anomaly*. Thus, a user must specify only m2, or specify m1, m2 and m3 in the visible condition.

To detect such anomaly at compile time, the authors propose a view checking mechanism in the model. When a method in a source program is compiled, a compiler produces invoking method names in each method, in addition to the attributes described above, into a method directory in the database. That is, these attributes are invoking method names, a method name, a class to which it belongs, parameter types, the number of parameters, and the type of return value. When a view is created, these attributes are propagated from the source classes. If a visible condition is specified for the view, the view is checked with the following procedure. First, a list v', which consists of method names

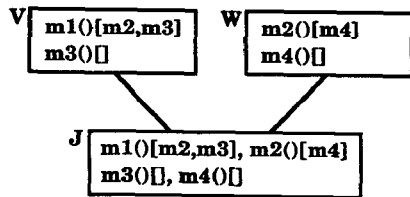


Figure 6: Propagation of Method Attributes

in a visible condition and their invoking method names, is created. Secondly, if v is a list of method names in the visible condition, whether or not v is equivalent to v' is checked. If the check shows that the equivalency is true, a new join view class definition is permitted. For example, in Fig. 6, when classes V and W are joined, all attributes of methods for both classes are propagated into the join view class J. If the visible condition for the join is specified with

```
view m1, m2 ;
```

then an expansion of the list is ($m1, m2, m3$). Thus, the view class definition is in error, because two lists do not match. However, if the visible condition is

```
view m2, m4 ;
```

then the expansion of the list is also ($m2, m4$). Thus, the view class definition succeeds, because the list in the visible condition matches with an expansion list.

4 Update Problems in the Object-Oriented Data Model

In the author's object-oriented data model, update operations through a view class are directly accomplished for the conceptual class instances. In a relational data model, however, the view operations result in copies into a workspace. Therefore, data update should be accomplished, both for the view relation and for base relations. This causes complicated semantic update problems [Kell86].

Other problems arise, when update operations are applied to a view class, whose construction involves select or join operations. The remaining portion of this section discusses how to resolve such problems for selection operation.

The view class "Module_for_Debug" example, as indicated in Section 2.1, is considered here. The conceptual class "Module" and its instances are shown in Fig. 4. If a function, named "Debug_PrintTree", becomes unnecessary, in order to release the program, a programmer may request deletion of the object "Debug_PrintTree" from the view class "Module_for_Debug". In this case, two different methods can be considered to update the object. One way is that the object "Debug_PrintTree" should be deleted from the Module class. This means that the function "Debug_PrintTree" is eliminated from the source program. Another way is, first, to exclude the object "Debug_PrintTree" from the view class (clear a pointer in the <instance-set>), and second to replace the value "Debug_PrintTree" in the instance variable

"name" with a value "PrintTree". This means that the function becomes unnecessary for the programmer, but continues to work for other programmers. In relational data model, each user must translate the deletion of a tuple in the view "Module_for_Debug" into the deletion or the replacement of a tuple in the base relation "Module" according to his own specific intentions. This translation should be specified in the user's application program. However, in the object-oriented data model presented here, a user or a database administrator can define a specific method "deletion" for the view class "Module_for_Debug", which includes deletion or replacement procedure for the conceptual class.

As mentioned above, by defining a view class and methods, users can store the update semantics into databases. Therefore, an individual user need not describe the update procedure in his own specific application program. As a result, application programs become simple, and incorrect updates can easily be excluded.

5 Conclusion

An object-oriented data model, which incorporates a new view mechanism, has been presented. A view class to realize the view mechanism has been proposed. Some problems have also been resolved, such as method hiding (i.e. method projection) and direct update, by the introduction of the view class concept. The data model includes distinct mechanisms, such that an instance not only belongs to one conceptual class but to several view classes, method selection when sending a message to an instance, and the query language for manipulating view classes. On the other hand, introducing method projection instigates a new problem, called view anomaly. To solve this problem, the authors proposed a view checking mechanism at compile time.

The view class mechanism, proposed in this paper, is currently being implemented in an object-oriented database management system, called "Odin", at NEC C&C Systems Research Laboratories.

References

- [ABDD89] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D. and Zdonik, S. "The Object-Oriented Database System Manifesto," *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases*, 1989, pp.40-57.
- [Kell86] Keller, A.M. "The Role of Semantics in Translating View Updates," *IEEE Computer*, Jan. 1986, pp.63-73.
- [ShZd89] Shaw, G.M. and Zdonik, S.B. "Object-Oriented Queries: Equivalence and Optimization," *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases*, 1989, pp.264-278.
- [TaCh89] Tanaka, K. and Chang, T-S. "On Natural Joins in Object-Oriented Databases," *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases*, 1989, pp.279-293.
- [TaYI88] Tanaka, K., Yoshikawa, M. and Ishihara, K. "Schema Virtualization in Object-Oriented Databases," *Proc. Int. Conf. on Data Engineering*, 1988, pp.23-30.