NORTH-HOLLAND

Informatics and Computer Science

An Object Model at Conceptual Level to Support Updatable Views on Object-Oriented Databases

WEN-WEI PAN

and

WEI-PANG YANG*

Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, R.O.C.

ABSTRACT

The research on view mechanisms for object-oriented databases can be classified into two independent issues, languages to define views and the underlying object models. This paper presents an object model at the conceptual level of abstraction. We call it the view class and real-world object model (V–R model). The major goal of the V–R model is intended to support updatable views efficiently for the object-oriented database systems. The V–R model elaborates a special method to describe the instance-of relationship between stored objects and classes so that a stored object can be the instance of several classes simultaneously. Query results and views can be managed like classes without creating temporary objects or tuples since the V–R model is closed against query operations. In this way, updates on views can be supported easily.

This paper also explores the update semantics in depth. By utilizing the specialty of the V–R model, a more eligible update translation is proposed for the updates on classes with or without inheritance relationships and for the updates on views. To show that the V–R model is not only theoretical, how to implement the V–R model and how to integrate the V–R model into an object-oriented database system are illustrated. ©*Elsevier Science Inc. 1996*

1. INTRODUCTION

The view mechanism has been very successful in relational databases (RDBs). It can hide certain data from users for security, or help to create

*Corresponding author.

© Elsevier Science Inc. 1996
655 Avenue of the Americas, New York, NY 10010

0020-0255/96/$15.00
PII S0020-0255(96)00125-9

derived relations that users are interested in. Views are intended to behave like relations; however, not all views can be updated as relations can. Views basically are defined by applying powerful relational operations, and they normally do not store data as relations do [2, 9]. An update to the view must be translated into an update to the actual relations in the conceptual model of the database. Reference [5] points out that a view can be updated only if it preserves base relations' keys. Therefore, the view update problem greatly lowers the utilization and objective of views.

Much research [1, 8, 10, 11, 15, 17] discusses the view mechanism in object-oriented database (OODBs), and we conclude two independent topics from it. The first topic concerns the way to define views. A common structure of view definition for OODBs contains the *type definition* and the *instance population*. The type definition describes the attributes and methods of a view. The instance population describes what objects are the instances of a view, and it usually is a query. In the work of [1], the definition of views allows users to organize views into the *virtual class hierarchies*, so that the roles that views play in the virtual class hierarchies are the same as classes do in the class hierarchies.

The second topic concerns the object model underlying the query language. The object model defines the structures and management of the stored objects as well as classes. The *query's results* and *view updates* are two major issues closely related to views. To manage query's results, the flat tuple is the simplest approach [8]. Each tuple's attribute is atomic, that is, *printable*. Flat tuples are not stored in the database, but exist temporarily for viewing. They do not have *object identifiers* (OIDs) and their lifetimes are typically limited to a transaction. The drawback of flat tuples is that users cannot query upon them anymore since their structures are different from stored objects. To overcome the flat tuple's drawback, [1] proposes the idea of *imaginary objects*. Each tuple can be converted into an imaginary object by creating a new OID for itself. Therefore, views can be populated with imaginary objects as classes can be populated with objects. However, one subtle problem may arise while OIDs being created. Consider two SQL-like queries:

Q1: *select F from Family where F.size* > 5 *and F.Father.Age* < 25

Q2: *select F from Family where F.size* > 5 *and F in* (*select F from Family where F.Father.Age* < 25)

With the first query, we obtain as many objects as families that satisfy the criteria. With the second query, the result is implementation-dependent, and we may obtain an empty set if new OIDs are generated for imaginary objects. The solution is to assign the same OIDs to those imaginary objects representing the same real world things. Users must be very careful to

choose the *core attributes* of views to do that, especially when a view is derived from several classes.

References [11, 15] suggest another approach to manage query results. The instances of a view are *materialized*, but not saved in the storage system. Materialized instances have the same data structure as stored objects. A hashing table is built to establish a one-to-one mapping between the materialized instances of a view and the stored objects of the class from which the view is derived. This hash table not only helps to maintain data consistency between stored objects and materialized instances, but also provides the tracking of materialized instances' OIDs. Nevertheless, users must specify the *OID-function* to make the one-to-one mapping exist. This idea, actually, is the same as that of core attributes; hence, it suffers the same problem.

In the work of [17], *object preserving operators* in the object model *COCOON* is proposed. In COCOON, an object can be the instance of several objects simultaneously, which is different from the previous two works. The query result is a set of existing objects, instead of creating objects. This approach avoids duplicating data and saves the cost of creating objects. Furthermore, view updates can be performed on existing objects directly. Consider this example: a view *EmpSal* is defined as "*create view EmpSal (Name, Salary) as (select Name Salary from Employee)*" and the class *Employee* has only one key, the *ID#*. Apparently, *EmpSal* is not updatable in RDBs because it does not preserve *Employee*'s key. For the OODBs with the object creating approach, *EmpSal* is updatable if a hashing table or some mechanisms else build up the one-to-one correspondence. In COCOON, *EmpSal* will contain the same set of stored objects as *Employee* does. So, update operations on *EmpSal* can be performed on the stored objects. Reference [14], however, argues that the database system will suffer the performance penalty while allowing an object to be the instance of several classes. One of difficulties is how to efficiently model the *instance-of relationship* such that an object can be the instance of several classes. COCOON, on the other hand, does not elaborate a concrete solution.

An essential topic, *the semantics of updates*, is neglected in all previous works. Suppose that we delete a student from the class *Student* that inherits the class *Person*. Does this operation mean that the student is removed from the database or the student is no more a student but still a person? This problem arises in OODBs because object-oriented data models capture the inheritance relationship between classes. Consequently, the update semantics becomes ambiguous. In fact, the update semantics for views are even more ambiguous in OODBs. Therefore, it is

essential to study the update semantics for classes with or without inheritance relationships and for views in OODBs.

In this paper, we propose a framework of the object model at the conceptual level of abstraction, called *the view class and real-world object model* (V–R model). The major goal is to support updatable views effectively. The V–R model elaborates a special method to describe to instance-of relationship, so that an object can be the instance of several classes and views simultaneously without performance reduction. The query's results and updates of views can be manipulated easily by this approach. We also address the implementation techniques about the V–R model to show that the V–R model is not only theoretical. Besides, based on the V–R model, we discuss the update semantics for OODBs thoroughly.

The organization of this paper is as follows. In Section 2, we introduce the basic concepts and formal definitions of the V–R model. We illustrate how an object can be the instance of many classes. We also prove that essential query operations are closed in the V–R model. In Section 3, the update semantics in the V–R model are discussed. We first clarify the update semantics, and then we define the view update translation based on the V–R model. Since views in the V–R model can be materialized, we present an efficient approach to keep materialized views up to date. In Section 4, we illustrate how to integrate the V–R model into an OODB system, and suggest two possible approaches to implement the V–R model. Finally, we give the conclusions in Section 5.

2. THE VIEW CLASS AND REAL-WORLD OBJECT MODEL (V–R MODEL)

The original idea of the V–R model comes from the real world. In the real world, people may see different aspects of an entity from different viewpoints. When the entity's information varies, all the people can perceive the changes. In the V–R model, people, viewpoints, and entities correspond to users, view classes, and real-world objects, respectively. Real-world objects denote concrete or abstract entities in the real world and convey entities' information, while view classes are viewpoints to observe real-world objects. For example, in Figure 1, users observe three different *images* of the real-world object *Jack* via three view classes. No matter how many view classes are created to observe *Jack*, his information is not duplicated. Consequently, all the view classes that observing *Jack* can see changes made to *Jack* automatically.

Note that *Jack* is perceptible via three classes, but *Mary* is only perceptible via the view class *Citizen*. The function of the perceptibility

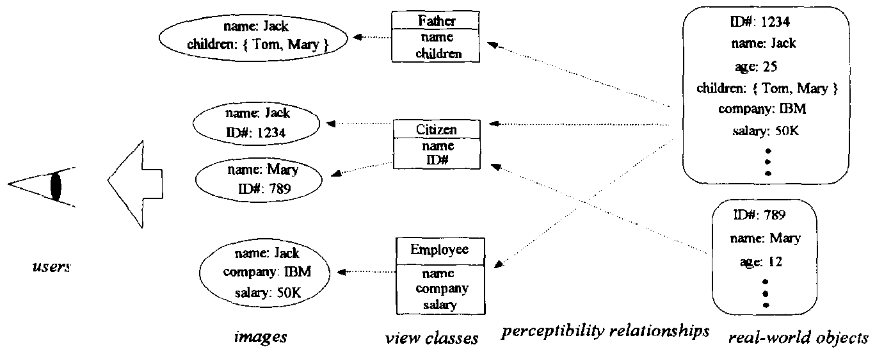

Fig. 1. The basic concepts on the V–R model.

relationship is to describe which real-world objects and what portions of their information are perceptible to the view class, which is more than the function of the instance-of relationship that only states the former.

DEFINITION 1. A real-world object (RO) is an entity, either concrete or logical, in the real world. It is represented as a tuple $\langle \mathbf{I}, \mathbf{D} \rangle$, where

1. **I** denotes the unique real-world object identifier (RID).
2. **D** denotes the stored data and is define as a set of properties. A property is defined as "$P : V$," where P is the property name and V denotes its value. A value is a single or a set of integers, real number, strings, and RIDs.

Definition 1 describes the RO's conceptual notation which captures three essential features of OODBs [4, 13, 14], the unique object identifiers, complex objects, and set values. Methods are omitted in Definition 1 because they can be treated as stored data as properties. Besides, *composite objects*, one feature of OODBs, do not appear in Definition 1. From the discussion of [12], composite objects result from applying a sort of integrity constraints. Therefore, it is the query processor's responsibility to support them.

ROs serve as the data source in the V–R model, and convey their own data. They are assumed to be stored on the permanent storage device, and their RIDs are the access keys to retrieve specific ROs.

DEFINITION 2. A view class C is represented as a tuple $\langle \mathbf{S}, \mathbf{N}, \mathbf{T}, \mathbf{E} \rangle$, where

1. **S** denotes the superclass of C. It is a set of view classes and can be empty.
2. N denotes the class name.

3. **T** denotes the intension. It is a set of attributes. An attribute is denoted as "$A:D$," where A is the attribute name and D denotes the domain of the attribute that can be integers, real numbers, strings, and view classes.

4. **E** denotes the extension. It is a set of conceptual objects (CO). Each conceptual object is represented as a tuple $\langle (r, vf) \rangle$, where r is an RID, vf is the view function that is defined as "vf: $\mathbf{D} \rightarrow \mathbf{T}$," $\mathbf{D}$ denotes RO's stored data, and vf is an onto function.

5. For each view class C_i in **S**, C_i's intension is included in or equal to **T**.

Definition 2 describes the conceptual notation of view classes. A view class has *intensional* and *extensional* notions [4]. The intensional notion corresponds to the definition of an abstract data type in object-oriented programming. It defines a viewpoint to "observe" ROs. The extensional notion serves as the basis for formulating queries. COs play the role of the perceptibility relationships illustrated in Figure 1. The *mapping pair* (r, vf) in a CO describes which properties of the RO with RID r are perceptible in the view class. A real object O_R is perceptible to a view class C if and only if O_R's stored data can be mapped onto C's intension since the view function is defined to be an onto mapping. For example, *Mary* in Figure 1 can be mapped onto the view class *Citizen* only.

The ways to compute a view class's extension depend on how the V–R model is implemented, and will be discussed in Section 4. We illustrate an example in Figure 2 which shows a conceptual database by the V–R model. There are three view classes and seven ROs. Arrow lines denote

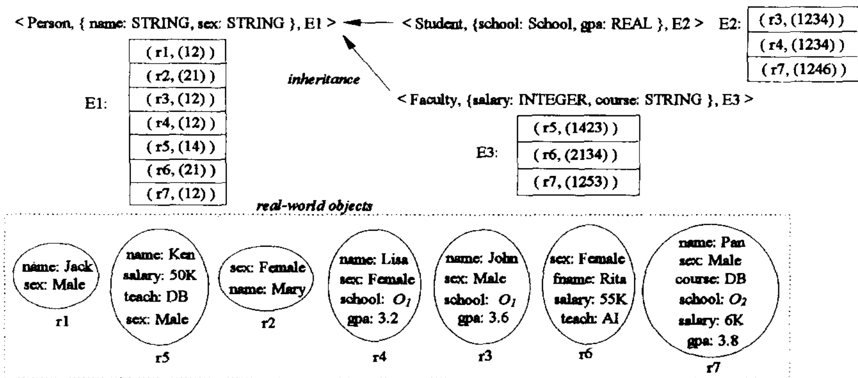

Fig. 2. A simple database presented by V–R model.

inheritance relationships and point to superclasses. They organize view classes into *class hierarchies*. The class extensions are presented as tables, and the view functions are denoted by vectors. The ith number x in a vector indicates that the xth property of an RO is mapped to the ith attribute of a view class. For example, the CO ($r7$,(1253)) in E3 will present the image "*name: Pan, sex: Male, salary: 6K, course: DB.*"

Note that the RO $r7$ is perceptible to the view class *Student* and *Faculty*. The V–R model allows the RO to contain more information than the view classes in a specific class hierarchy. It is not necessary to create a subclass *TA* of both *Student* and *Faculty*.

If an RO is perceptible to the view class $C1$ and C is the superclass of $C1$, then the RO must be perceptible to C. This property results from the inheritance relationship.

LEMMA 1. *If the view class $C1$ inherits C, then for each conceptual object $\langle(r,vf)\rangle$ in $E(C1)$, there exists a conceptual object $\langle(r',vf')\rangle$ in $E(C)$ such that r is equal to r'.*

The proof of Lemma 1 is straightforward. The view class *Person*, *Student*, and *Faculty* in Figure 2 can illustrate this property.

The conceptual notion of views is very similar to that of view classes. Views, actually, have the same functionality as view classes. They both provide users with various viewpoints to observe ROs.

DEFINITION 3. A view C is represented as a tuple $\langle \mathbf{N},\mathbf{T},\mathbf{Q},\mathbf{E}\rangle$, where

1. **N** denotes the view name.
2. **T** denotes the intension. Its representation is the same as the view class's intension.
3. **Q** is the instance population. It is a query.
4. **E** denotes the extension. It is a set of conceptual objects. Each conceptual object is represented as a tuple $\langle(r_1,vf_1),(r_2,vf_2),\ldots,(r_n,vf_n)\rangle$, where $n \geqslant 1$, r_i is an RID, and vf_i is the view function. Let $vf = vf_1 \cup vf_2 \cup \cdots \cup vf_n$; then vf will be an onto function.

The CO of a view may contain several mapping pairs. The number of mapping pairs in a CO, the RO's retrieval speed, and the implementation of the view function will determine the speed of computing a CO's image. A view's extension is the result of its instance population, a query. It normally is not stored in the storage system as the view class's extension. However, users sometimes may want to store a view's extension for the performance. A view is called *semi-materialized* if its extension is stored in the storage system.

Since the instance population of a view is a query, we need to prove an essential property, the closure property, for Definition 3. The closure property ensures that the result of query operations can be expressed as a set of COs defined in Definition 3. In the following, we show how the view's intension and extension can be computed and expressed as a set of COs for each operation. Some shorthand notations are defined first. Let o be a CO in a view class C or in a view V; then

- $Rids(o) = \{r_1, r_2, \ldots, r_n\}$ and $vf(o) = vf_1 \cup vf_2 \cup \cdots \cup vf_n$.
- $M(r_i)$ denotes the stored data of the RO with RID r_i.
- $M(o)$ denotes all the RO's stored data that o refers to. They are equal to $\bigcup_i M(r_i)$.
- $IM(o)$ denotes the image presented by o. It is equal to $\bigcup_i vf_i(M(r_i))$.
- $IM(o).Attr$ denotes the value of the attribute $Attr$ in the image.
- $E(V)$ denotes the extension of V, and $T(V)$ denotes the intension of V.

Let V_i and V_j, view classes or views, denote the operands of operations and V denote the resulting view of the operation.

(*a*) ***Restriction***: $V = \sigma_{Preds}(V_i)$, where *Preds* denotes the restriction predicate. Then
  - $T(V) = T(V_i)$.
  - For each conceptual object $o \in E(V_i)$, if $IM(o)$ satisfies *Preds*, add o to $E(V)$.

(*b*) ***Projection***: $V = \pi_{Attrs}(V_i)$, where *Attrs* denotes the projected attributes. Then
  - $T(V) = \{Attrs\}$.
  - For each conceptual object $o \in E(V_i)$, add o' to $E(V)$, where $Rids(o') = Rids(o)$ and $vf(o') = v'$, where v': $M(o) \rightarrow T(V)$.

Since V's intension is a subset of V_i's the resulting CO may contain one or more mapping pairs whose view functions are null. In other words, those mapping pairs do not constitute the CO's image. The reason for keeping such mapping pairs in a CO is to establish a one-to-one correspondence for the view updates. We will explain this later.

(*c*) ***Union***: $V = V_i \cup V_j$. Then
  - $T(V) = T(V_i) \cap T(V_j)$.
  - For each $o \in E(V_i)$ or $o \in E(V_j)$, add o' *to* $E(V)$, where $Rids(o') = Rids(o)$ and $vf(o') = v'$, where v': $M(o') \rightarrow T(V)$.

The requirement of a legal union operation in the V–R model is that the intensions of V_i and V_j are *compatible*; that is, they have common at-

tributes. Note that the intension of the resulting view is the intersection of those of V_i and V_j.

(d) ***Difference***: $V = V_i - V_j$. Then
- $T(V) = T(V_i)$.
- For each $o \in E(V_i)$, if $\forall\ o' \in E(V_j) \wedge Rids(o') \neq Rids(o)$, add o to $E(V)$.

The V–R model makes use of RIDs to determine whether two COs are equal or not. This approach, called *RID-equal*, is different form using CO's images, which correspond to using instance's values, called *value-equal*, in traditional RDBs and OODBs. Suppose the view class *Person* does not have a key in Figure 2; then it is ambiguous to determine whether the CO $(r7,(1253))$ in *Faculty* differs from the CO $(r7,(1246))$ in *Student* by the traditional approach with the operation $(Student - Faculty)$. However, in the V–R model, we can get rid of $(r7,(1246))$ in the result since they refer to the same real-world entity. Thus, the difference operation will not suffer the ambiguity even if V_i and V_j do not have a key in common. Actually, the RID-equal is conceptually the same as the *referential-equal* [16] that tests whether two database objects refer to the same real-world entity. The V–R model can directly support it instead of using methods to simulate it.

(e) ***Cartesian product***: $V = V_i \times V_j$. Then
- $T(V) = T(V_i) \cup rename(T(V_j))$.
- For each $o \in E(V_i) \wedge o' \in E(V_j)$, add o'' to $E(V)$, where $Rids(o'') = Rids(o) \cup Rids(o')$ and $vf(o'') = vf(o) \cup vf(o')$.

The "*rename*( )" denotes the renaming operation that will make $T(V_j)$ different from $T(V_i)$.

(f) ***Natural join***: $V = V_i \bowtie V_j$. Let $Attrs = T(V_i) \cap T(V_j)$; then
- $T(V) = T(V_i) \cup T(V_j)$.
- For each $o \in E(V_i) \wedge o' \in E(V_j)$, if $IM(o).Attr = IM(o').Attr$, add o'' to $E(V)$, where $Rids(o'') = Rids(o) \cup Rids(o')$ and $vf(o'') = vf(o) \cup vf(o')$

The natural join can be expressed by a projection, a restriction, and a Cartesian product: $V_i \bowtie V_j \equiv \pi_{T(V_i) \cup T(V_j)}(\sigma_{V_i \cdot Attrs = V_j \cdot Attrs}(V_i \times V_j))$.

(g) ***Navigation***: $V = V_i.Attr$. *Attr* is V_i's nonatomic attribute whose domain is the view class C. Then
- $\mathrm{T}(V) = \mathrm{T}(C)$.
- For each $o \in E(V_i)$, let $r' = IM(o).Attr$, add o' to $E(V)$, where $Rids(o') = Rids(o) \cup \{r'\}$ and $vf(o') = v'$, where v': $M(o') \rightarrow T(C)$.

The navigation operation also can be expressed by a projection, a restriction, and a Cartesian product: $V_i.Attr \equiv \pi_{Attr}(\sigma_{V_i.Attr = C.RID}(V_i \times C))$.

(h) ***Intersection***: $V = V_i \cap V_j$. Then

- $T(V) = T(V_i) \cap T(V_j)$.
- For each $o \in E(V_i)$, if $\exists o' \in E(V_j) \wedge Rids(o') = Rids(o)$, add o'' to $E(V)$, where $Rids(o'') = Rids(o)$ and $vf(o'') = v$, where v: $M(o'') \rightarrow T(V)$.

The intersection operation can be expressed by one union and two difference operations: $V_i \cap V_j \equiv V_i - ((V_i \cup V_j) - V_j)$.

Apparently, the result of query operations can be expressed as a set of COs defined in Definition 3. Hence, the V–R model is closed under these query operations. Furthermore, two conclusions can be obtained from the discussion. First, there always exist a one-to-one mapping between each CO of the view and a corresponding set of COs of operands on which the view is defined. The reason is that for each operation, the mapping pairs of operands' COs are kept in the corresponding CO of the resulting view. This property is essential to make a view updatable. A brief algebraic explanation is as follows. A view's definition, in general, can be expressed as $\pi_{Attrs}(\sigma_{Preds}(T_1 \times T_2 \times \cdots \times T_n))$, where T_i denotes a view class, the union of view classes, or the difference of view classes [5]. According to the operation (c) and (d), all the COs in each T_i contain only one mapping pair. Then a CO in the view will be $\langle (r_1, vf_1), (r_2, vf_2), \ldots, (r_n, vf_n) \rangle$ according to the operation (a), (b), and (e), where each mapping pair corresponds to a CO in T_i for $1 \leqslant i \leqslant n$.

Second, projections, unions, differences, and Cartesian products only need to manipulate class extensions to calculate the extension of the view. So, it is not necessary to retrieve ROs' stored data to compute a view's extension if a view's definition only involves these four operations.

3. UPDATES IN THE V–R MODEL

Three update operations, *create*, *delete*, and *modify*, are considered in the V–R model. They can be performed on ROs, view classes, and views; for simplicity, we call them RO updates, view class updates, and view updates, respectively. Since ROs possess stored data, RO updates can be performed directly without doubt. Therefore, we will concentrate on the view class and view updates in this section.

Two separate processes will occur while performing the view class or view updates. The first one is the translation of the view class or view updates into RO updates because only ROs possess stored data. The

second process is the extension maintenance for the view classes and semi-materialized views due to ROs' changes. In the following, we first clarify the semantics of view class and view updates in Section 3.1, and then we define the update translation in Section 3.2 according to the clarified semantics. Finally, we introduce an efficient way to do the extension maintenance in Section 3.3.

3.1. *THE SEMANTICS OF VIEW CLASS AND VIEW UPDATES*

3.1.1. *View Class Updates*

Since the CO in a view class contains only one mapping pair, view class updates clearly should be performed on the referred RO. However, the semantics of *create* and *delete* are not clear. Consider a creation on the view class *Faculty* in Figure 2: "*create*⟨*name*: *John*, *sex*: *Male*, *salary*: $6K$, *teach*: X⟩*into Faculty*." If the view class *Person* does not have a key, then a new RO can be created and the database will contain two ROs having the same name and sex, *John* and *Male*, respectively. Otherwise, this operation should be rejected because the RO $r3$ already exists in the database and is a student. The V–R model introduces the notion of *object evolution* that can gracefully deal with the second condition instead of rejecting it. Object evolution means that an RO may change the amount of its stored data. As for this example, the RO *John* will be extended to have the extra stored data, "*salary:* $6K$, *teach:* X." Formally, suppose C is a view class in the class hierarchy H, and C_R is the root of H. When performing a creation on C, the system needs to check the condition: "$\exists\ O \in E(C_R), IM(O).Key = IM(O').Key$," where O' is the CO to be created and *Key* is one of C_R's candidate keys. If this condition stands, then the RO referred in O should be extended.

On the other hand, consider a deletion on the view class *Student*: "*delete from Student where name = Pan*." Does this operation mean that the student *Pan* is removed from the database or he is no longer a student? To proceed with the idea of object evolution, the RO *Pan* should be shrunk to drop the stored data, "*school:* O_2, *gpa:* 3.8." Note that it is not necessary to actually delete those stored data from the database to achieve it.

Object evolution will cause some changes to the class extensions. For instance, when an RO is extended to a view class, a new CO must be added to that view class.

3.1.2. *View Updates*

The semantics of view updates is more ambiguous. Figure 3 shows an EMPLOYEE database. It contains three view classes, $\langle \emptyset, Emp, \{name\colon String,\ dept\colon Dept\}, E1\rangle$, $\langle \emptyset, Manager, \{name\colon String,\ manage\colon Dept\}, E3\rangle$, and $\langle \emptyset, Dept, \{name\colon String,\ location\colon String\}, E2\rangle$. The "$ID_x$" and "$ID_y$" are the object identifiers of two department instances. The view *EM* is defined as $\pi_{Emp.name, Manager.name}(\sigma_{Emp.dept=Manager.manage}(Emp \times Manager))$. Consider the update $u1$ in Figure 3. Does it mean to remove *John* and *Mary* out of the database or only break the association imposed upon them? Reference [15] claims to remove the *root object* of the navigation path only to handle the deletion on views. This approach only works for those views derived from the navigation operation. As this example, there is no root object because the view *EM* is a join view. The V−R model adopts the latter, breaking the imposed association, to handle the deletion on views. Therefore, either John's attribute *dept* or Mary's attribute *manage* should become null value. Users have the responsibility for choosing the appropriate one.

In addition to clarifying the view update semantics, we have to revise the notion of a correct view update translation that proposed for RDBs [2, 9]. Suppose the view f is updated by a view update u; then the database must instead be updated by a database update T. Consequently, T is a *translation* of u if and only if the following constraints stand.

(1) $u \circ f(s) = f(T \circ s)$, where s denotes a database state. (*consistent constraint*)

(2) $\forall\ s, u \circ f(s) = f(s)$ then $T \circ s = s$. (*acceptable constraint*)

In these two constraints, a view is treated as a mapping from the database status to the view scheme. The first one indicates that T takes the database to a state that maps onto the update view. Is every consistent T acceptable? Consider a database update "*Create*$\langle$*Joe*, $ID_z\rangle$*into Manager.*" It is consistent with $u2$ in Figure 3, but not acceptable. It changes the database, although no changes are made in the view. Therefore, the second constraint makes sure that T will not change the database if u

name	dept
John	IDx
Frank	IDx
Rita	IDy

Emp

name	manage
Mary	IDx
Acer	IDy

Manager

name	location	
H.Q.	L.A.	IDx
E.C.	N.Y.	IDy

Dept

name	name
John	Mary
Frank	Mary
Rita	Acer

EM

u1: delete <John, Mary> from EM.
u2: modify <Peter, Jane> to <Peter, Mary> in EM.
u3: modify <John, Mary> to <John, Linda> in EM.

Fig. 3. The EMPLOYEE database.

does not change $f(s)$. However, this notion of view update translation is too restricted sometimes. Consider the view update $u3$ in Figure 3. It means to change *John*'s department manager to *Mary*; then a reasonable database update T' will be "*Change*$\langle$*Mary*, $ID_x\rangle$*to*$\langle$*Mary*, *null*$\rangle$" and plus "*Create*$\langle$*Linda*, $ID_x\rangle$*into Manager*." T' actually means to break the association; nevertheless, it is not consistent with $u2$ because the entry $\langle$*Frank*, *Mary*$\rangle$ in *EM* will also be changed to $\langle$*Frank*, *Linda*$\rangle$.

In conclusion, to establish or to break the association between the ROs in a CO is the fundamental semantics for view updates. In the following, we propose a new view update translation on this basis.

3.2. *VIEW UPDATE TRANSLATION*

From the discussion at the end of Section 2, a view V's definition can be expressed as $\pi_{Attrs}(\sigma_{Preds}(T_1 \times T_2 \times \cdots \times T_n))$, and its CO is $\langle(r_1, vf_1), (r_2, vf_2), \ldots, (r_n, vf_n)\rangle$. The *Preds* denotes a Boolean predicate that enforces restrictions either upon individual mapping pairs or between two of them. We call these two kinds of restrictions the *individual restrictions* and the *associated restrictions*, respectively. The associated restriction, furthermore, is either a navigation restriction, expressed as "$IM(\langle(r_x, vf_x)\rangle).a_1 = r_y$," or a join restriction, expressed as "$IM(\langle(r_x, vf_x)\rangle).a_1 \sim IM(\langle(r_y, vf_y)\rangle).a_2$," where "$\sim$" denotes a comparator; a_1 and a_2 are attributes.

The translation of a deletion on a view is to break the associated restrictions imposed on a CO. To simplify the discussion of view update translation, we assume that all associated restrictions are conjunct. Consequently, we only need to break one of the associated restrictions in *Preds* to accomplish the translation. Users have the responsibility to indicate which associated restriction r in *Preds* to break in the deletion operation. Suppose the indicated associated restriction r is a join restriction expressed as above; then a null value is assigned to either $M(r_x).vf_x^{-1}(a_1)$ or $M(r_y).vf_y^{-1}(a_2)$. Otherwise, r is a navigation restriction; then a null value is assigned to $M(r_x).vf_x^{-1}(a_1)$ without doubt. The shorthand "$M(r_x).vf_x^{-1}(a_1)$" indicates $M(r_x)$'s property to which the attribute a_1 is mapped by vf_x.

The translation of a modification on a view will depend on the changes of the images. Suppose two mapping pairs, $\langle r_x, vf_x\rangle$ and $\langle r_y, vf_y\rangle$, in a CO contain parts of images that will be modified; then we may end up with four possible conditions. We list each of them and their corresponding translations below.

- $\langle r_x, vf_x\rangle$ and $\langle r_y, vf_y\rangle$: Both mappings pairs still refer to the original ROs; then the modifications on $IM(\langle(r_x, vf_x)\rangle)$ and $IM(\langle(r_y, vf_y)\rangle)$ are performed on $M(r_x)$ and $M(r_y)$, respectively.

- $\langle r'_x, vf'_x \rangle$ and $\langle r_y, vf_y \rangle$: One of the mapping pairs refers to a new RO; then the following steps are executed if one or more associated restrictions exist between $IM(\langle r_x, vf_x \rangle\rangle)$ and $IM(\langle(r_y, vf_y)\rangle)$:
  1) Given an associated restriction p between $\langle r_x, vf_x \rangle$ and $\langle r_y, vf_y \rangle$.
  2) If p is a join restriction, then perform

$$\text{“}M(r_x).vf_x^{-1}(a_1) \leftarrow null\text{”} \quad \text{and}$$

$$\text{“}M(r'_x).vf_x^{-1}(a_1) \leftarrow M(r_y).vf_y^{-1}(a_2).\text{”}$$

  3) Otherwise, if p is “$IM(\langle(r_x, vf_x)\rangle).a_1 = r_y$,” then perform

$$\text{“}M(r_x).vf_x^{-1}(a_1) \leftarrow null\text{”} \quad \text{and} \quad \text{“}M(r'_x).vf_x^{-1}(a_1) \leftarrow r_y.\text{”}$$

  4) Otherwise, if p is “$IM(\langle(r_y, vf_y)\rangle).a_2 = r_x$,” then perform

$$\text{“}M(r_y).vf_y^{-1}(a_2) \leftarrow r'_x.\text{”}$$

- $\langle r_x, vf_x \rangle$ and $\langle r'_y, vf'_y \rangle$: It is symmetrical to the above one.
- $\langle r'_x, vf'_x \rangle$ and $\langle r'_y, vf'_y \rangle$: Both of the two mapping pairs refer to new ROs; then a deletion is performed to break the association between $\langle r_x, vf_x \rangle$ and $\langle r_y, vf_y \rangle$.

The modifications discussed above are assumed not to create new COs. If they do, the above processes should be performed after creating new COs. For example, the translation of $u2$ in Figure 3 will be as follows:

1. Create an RO of *Linda* in the database.
2. Insert a CO, $\langle r_{Linda}, vf_{Linda} \rangle\rangle$, into the $E(Manager)$.
3. Assign *null* to $M(r_{Mary}).vf_{Mary^{-1}}(manage)$.
4. Assign $M(r_{John}).vf_{John^{-1}}(dept)$ to $M(r_{Linda}).vf_{Linda^{-1}}(manage)$.

Finally, the translation of a creation on a view is to insert a new CO with the format of $\langle(r_1, vf_1), (r_2, vf_2), \ldots, (r_n, vf_n)\rangle$. To do that, users need to create all referred ROs if necessary. After a new CO is constructed, all the restrictions imposed on the view should be validated.

3.3. *EXTENSION MAINTENANCE BY THE CHANGES OF ROS*

There are four conditions to maintain view classes' extensions, creating ROs, deleting ROs, evolving ROs, and modifying ROs. For clearness, the

derived views of a view class are those views whose definitions involve that view class. When a view class's extension is changed, all its superclasses and derived semi-materialized views must be updated recursively. The extension maintenance of the former is much simpler than that of the latter. In the following discussion, we mean semi-materialized views when addressing views.

DEFINITION 4. The view class C_T is the target class of the real-world object O_R if and only if O is perceptible to C_T, but not perceptible to all subclasses of C_T.

When an RO is created, new COs should be added to its target class C_T and all C_T's superclasses. The way to know an RO's target class depends on the implementation of the V–R model. The derive views of those updated view classes can be updated by the computation of a view's extension in Section 2. However, the following two views must be handled differently. Suppose the CO $\langle(r, vf)\rangle$ is added to $E(C)$; then

- if $V = C - C'$, then add $\langle(r, vf)\rangle$ to $E(V)$ while no COs in $E(C')$ refer to $RO(r)$.
- if $V = C \cap C'$, then add $\langle(r, vf')\rangle$ to $E(V)$ while there exists a CO in $E(C')$ referring to $RO(r)$. The vf' is the mapping: $M(r) \rightarrow T(V)$.

When an RO is deleted, all the COs referring to it should be removed. It will cost a great deal of time to search such COs. Therefore, we introduce a more efficient approach, *deferred-deletion*. Suppose that each RO has a *deletion mark* and a counter that records the number of COs that refer to it. When deleting a real-world object O_R, the system can remove it safely if its counter is equal to zero, or the system marks it instead. Those COs referring to O_R become *invalid*, but they will not be removed until the system performs the operations that need to retrieve O_R's stored data. At that time, the system can detect whether a CO is valid or not. While an invalid CO is found, it can be removed and the counter of referred RO is decreased by one. If the counter is equal to zero, then the RO can be removed safely. One subtle problem is that not all referred ROs in a CO need to be retrieved when computing the CO's images because view functions may be null, as we mentioned in Section 2. Therefore, such an invalid CO cannot be detected. To remedy this problem, we can mark all the derived views of O_R's target class such that they will be forced to check the deletion mark of all referred ROs in a CO when computing the CO's images. Marked views need to do so only once, and then their marks can be reset.

In the V–R model, only four operations need to retrieve ROs' data: displays, restrictions, joins, and navigations. In contrast, other operations

cannot detect invalid COs in deferred-deletion. Therefore, invalid COs may propagate when new views are derived from those operations. Figure 4 tabulates the propagation of invalid COs. For example, the last row indicates that the invalid CO remains in $E(V)$ only when both $E(C1)$ and $E(C2)$ contain the same invalid CO; otherwise, it will not propagate to $E(V)$. The Cartesian product will produce more invalid COs, but it is seldom used in queries alone. Navigations and joins are much more frequently used instead.

Object evolution also will add new COs or make existing COs invalid. The view update of creating ROs can handle the former case, and the idea of deferred-deletion can handle the latter case. We can validate the view function to detect invalid COs. Consider the example in Figure 2; while the student *Pan* is shrunk to be a faculty, the view function in $(r7,(1246))$ will become invalid because *Pan*'s fourth and sixth properties are no longer available.

In contrast to the previous three conditions, when the RO's stored data are modified, all the view classes remain unchanged, but some COs in views may become *inconsistent* with the constraints imposed on views. For example, a view *Adult* is defined as $\sigma_{age>=18}(Person)$. If we change a person's age from 20 to 15, then that person becomes inconsistent with *Adult*. On the contrary, if we change a person's age from 15 to 20, then that person becomes perceptible to *Adult*. The view update of creating RO can handle the latter case, and we adapt the deferred-deletion to detect inconsistent COs as follows. Let O_R be the modified RO and let C_T be O_R's target class. We can mark all the derived views of O_R's target class. Note that this mark is different from the deletion mark. Marked views are forced to evaluate their imposed constraints for each CO once when CO's image is computed.

Here is an example of a modification. The translation of $u2$ in Figure 3 is "$Create\langle Linda, ID_x\rangle into\ Manager;\ Change\langle Mary, ID_x\rangle to\langle Mary, null\rangle$." The former adds a CO to $E(Manager)$. Consequently, two COs, $\langle(r_{John},$

an invalid CO in →	*E(C1)* only	*E(C2)* only	Both
$V=\pi(C1)$	1	—	—
$V=C1\cup C2$	1	1	1
$V=C1-C2$	1	0	0
$V=C1\times C2$	card(*C2*)	card(*C1*)	card(*C1*)+card(*C2*)
$V=C1\cap C2$	0	0	1

Fig. 4. The propagation of invalid COs for different views.

$vf_{John}), (r_{Linda}, vf_{Linda})\rangle$ and $\langle(r_{Frank}, vf_{Frank}), (r_{Linda}, vf_{Linda})\rangle$, will be added to $E(EM)$ by the join operation. Their images are "*name: John*, *name: Linda*" and "*name: Frank*, *name: Linda*," respectively. The latter causes the view EM to be marked. When we want to display the content of EM, two inconsistent COs, $\langle(r_{John}, vf_{John}), (r_{Mary}, vf_{Mary})\rangle$ and $\langle(r_{Frank}, vf_{Frank}), (r_{Mary}, vf_{Mary})\rangle$, will be detected and removed.

4. IMPLEMENTATION

The V–R model can be implemented as a kernel in a OODB system. Figure 5 shows a simplified database's system architecture that incorporates the V–R model and two service interfaces. The view-class service interface provides the query processor with four kinds of operations to inquire and maintain the database. The RO service interface allows the V–R model to read, write, create, and delete the ROs in a storage system. The access key of ROs is the RID. The object-based storage systems, such as WiSS [7], can be used to implement the RO service interface.

The constructions of ROs, view functions, and extensions are essential to implement the V–R model. A flexible way of constructing ROs is the *typeless approach*, that is, each RO stores the format of its own stored data. The ROs in Figure 2 are typeless, for example. When creating an RO, users must determine its target class and the view function. The merit of the typeless approach is flexible since the format of stored data depends on the characteristics of individual ROs. For example, one object can store its picture in bitmap format, while another object can store its in the compressed picture format. The typeless approach is also good to dynamically add or delete RO's properties. However, its major disadvantage is the overhead to describe the format of properties in each RO.

An alternative to constructing ROs is the *typed approach*, that is, each RO belongs to a type. Assume that the management of types is supported. To create an RO, users simple select a type. One of the typed approach's merits is that the view function for a CO can be determined on the fly if

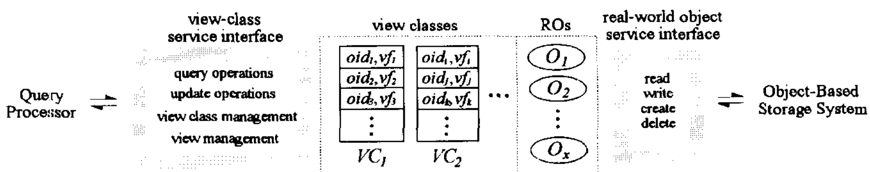

Fig. 5. Two service interfaces and the V–R model.

the system can know an RO's type either by the information stored on the RO or by some other mechanisms since the system can prepare the onto mapping from each type to each view class in advance. Figure 6 shows that same database as the one in Figure 2 by typed implementation. The types t_2 and t_3 inherit t_1, and t_4 inherits both t_2 and t_3. The mappings from the RO of t_4 to *Person*, *Faculty*, and *Student* will be (12), (1234), and (1256). A table lookup can find out the desired view function for a CO.

A variant of typed implementation is to make view classes serve as types. This approach prohibits an RO from having more attributes or behaviors than its target class. For example, the RO $r7$ in Figure 6 cannot exist unless a view class *TA* that inherits *Faculty* and *Student* exists in the schema.

The typed approach usually achieves better performance and requires less storage space than typeless one.

The extensions of view classes and views are the basis of performing query operation and update operations. Four kinds of processes performed on the extensions are: adding a CO, deleting a CO, processing each CO, and searching the COs that refer to a specific set of RIDs. The last process is necessary for differences and intersections. It is also useful if we want to ensure that each CO in an extension is unique. Moreover, it becomes feasible to immediately remove the invalid and inconsistent COs mentioned in Section 3 if this process is implemented fast enough. The nonorder array, the $B+$ tree, and the extensible linear hashing table are three major techniques to implement the extensions. We do not intend to study their advantages and disadvantages in this paper. Currently, we use the first one to program the V–R model, and the hashing index on RIDs

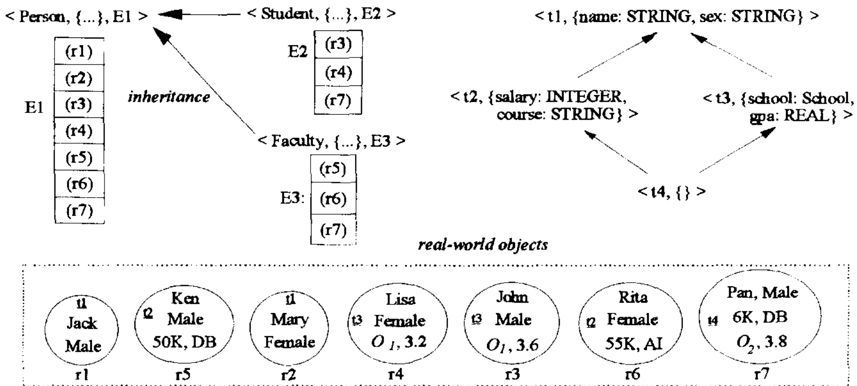

Fig. 6. Typed implementation for the V–R model.

will be dynamically built to obtain reasonable performance while the last process is performed.

5. CONCLUSIONS

In this paper, we present an object model, the V–R model, at the conceptual level of abstraction. The notions of real-world objects, view classes, views, and conceptual objects are defined formally. Based on the V–R model, we explore three important issues about the views. First, we prove that the V–R model is closed under the essential query operations. Second, we clarify the update semantics and formally define the view update translation for OODB systems. Third, we show how the V–R model can be integrated in an OODB system and discuss the V–R model implementation techniques.

We summarize the specific features of the V–R model and make a brief comparison with previous works on view mechanisms.

- The V–R model manages the derived objects which correspond to the conceptual objects at a low cost. It does not assign object identifiers for derived objects. Also, neither query results nor semi-materialized views duplicate stored data.
- The V–R model introduces a clear and complete solution for the update operations. The semantics of updates are different from traditional ways. Object evolution and the update translation make the update semantics more precise and reasonable.
- The extensions of view classes and views are different from the indices, such as nested indices and path indices [3]. An index provides a method of object retrieval based on some attributes for a special purpose. In the V–R model, the functions of extensions include the perceptibility relationship and the basis for performing query operation and update operations.

REFERENCES

1. S. Abiteboul and A. Bonner, Objects and views, in *Proc. ACM SIGMOD Conf. on Management of Data*, May 1991, pp. 238–247.
2. F. Bancilhon and N. Spyratos, Update semantics and relational views, *ACM Trans. Database Syst.* 6(4):557–575 (Dec. 1981).
3. E. Bertino and W. Kim, Indexing techniques for queries on nested objects, *IEEE Trans. Knowledge and Data Engrg.* 1(2):196–214 (Oct. 1989).
4. E. Bertino, M. Negri, G. Pelagatti, and L. Sbattella, Object-oriented query languages. The notation and the issues. *IEEE Trans. Knowledge and Data Engrg.* 4(3):223–237 (June 1992).

5. J. A. Blakeley, N. Coburn, and P.-A. Larson, Efficiently updating materialized views in *Proc. ACM SIGMOD Conf. on Management of Data*, May 1986, pp. 61–71.
6. J. A. Blakeley and N. L. Martin, Join index, materialized view, and hybrid-hash join: A performance analysis, in *Proc. IEEE 6th Int. Conf. on Data Engrg.*, Feb. 1990, pp. 256–263.
7. H. Chou, D. DeWitt, R. Katz, and A. Klug, Design and implementation of the Wisconsin storage system, *Software Practice and Exp.* 15(10) (Oct. 1985).
8. U. Dayal, Queries and views in a object-oriented data model in *Proc. 2nd Int. Workshop on Database Programming Language*, June 1989, pp. 80–91.
9. U. Dayal and P. Bernstein, On the correct translation of update operations on relational views, *ACM Trans. Database Syst.* 8(2):381–418 (Apr. 1988).
10. S. Heiler and S. Zdonik, Object views: Extending the vision, in *Proc. IEEE 6th Int. Conf. on Data Engrg.*, Feb. 1990, pp. 86–93.
11. M. Kifer, W. Kim, and Y. Sagiv, Querying object-oriented databases, in *Proc. ACM SIGMOD Conf. on Management of Data*, June 1992, pp. 393–402.
12. W. Kim *et al.*, Composite object support in an object-oriented database system, in *Proc. 2nd Int. Conf. on Object-Oriented Programming Syst., Languages, and Appl.*, Orlando, FL, Oct. 1987.
13. W. Kim, Object-oriented database: Definition and research directions, *IEEE Trans. Knowledge and Data Engrg.* 2(3):327–341 (Sept. 1990).
14. W. Kim, *Introduction of Object-Oriented Databases,* MIT Press, 1992, ch. 2.
15. W. Kim, *Modern Database Systems,* Addison-Wesley, 1995, ch. 7.
16. Y. Masunaga, Object identity, equality and relational concept, in *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases*, North-Holland, 1990, pp. 185–202.
17. M. H. Scholl, C. Laasch, and M. Tresch, Updatable views in object-oriented databases, in *Proc. 4th Int. Conf. on Deductive and Object-Oriented Databases*, North-Holland, 1992, pp. 189–207.

Received 22 June 1995; revised 22 November 1995, 29 February 1996