'89 KACC 1989. 10. 27~28

Properties of Object Composition Operations for Object-Oriented CAD Database Systems

Tae-Soo Chang[†] *Katsumi Tanaka*[‡]
[†]Graduate School of Science and Technology, Kobe University
Rokkodai, Nada, Kobe 657, Japan
[‡]Dept. of Instrumentation Engineering, Kobe University
Rokkodai, Nada, Kobe 657, Japan

ABSTRACT: In this paper, we introduce a recursively-defined *natural join* operation as well as well-known object composition operations (*union*, *intersection*) for composing CAD database objects. Then, we will discuss how to realize these operations by the message passing computing mechanism. Next, we will discuss what kind of behaviours (methods) are preserved under our natural join operations. Finally, we investigate mathematical properties about the relationships among several object composition operations (natural join, union and intersection).

1 Introduction

Recently, much attention has been focused on the use of Object-Oriented Database (OODB) Systems in CAD/CAM environment[1][2][3][4]. Since the notion of objects in OODB systems caputures both the static aspect (properties) and the behavioural aspect of objects, it is suitable for modelling several complex-structured objects appearing in CAD/CAM environment. Structurally, an object is constructed by applying tuple constructor operation and set constructor operation in a recursive manner. That is, a complex object might be a set of tuples whose element might be again a complex object. As for the behavioural aspects of objects, each behaviour of an object is represented by a procedure called *method*, and both of the static data and methods are encapsulated into an object.

Although the concept of OODB systems are regarded as useful for CAD databases and some commericial products appeared, it is not still clear what kind of operations are fundamental for those object-oriented CAD database systems. Especially, in CAD environment, operations for composing new objects from existing objects are very important, and must be supported in an efficient way by the system.

Section 2 gives motivating examples, intuitive explanation of our natural join operation and basic definitions. Section 3 gives how to realize natural joins by the message passing computing mechanism. Section 4 provides a way to obtain a set of preserved methods on the result of our natural join, based on the *method dependency graph*. Section 5 provides mathematical properties of composition operations. Section 6 is a concluding remarks.

2 Motivations and Basic Definitions

2.1 Generalization of Relational Natural Join Operators

In relational data model, the *natural join* combines two relations (sets of homogeneous tuples) and produces a bigger relation. In order to define a similar operator for complex objects, we believe that the following generalizations are necessary.

(a) Generalization of operand types
In object-oriented databases, an object is constructed by applying *tuple constructor* and *set constructor* in a recursive manner. For example, an object might be a tuple whose attribute values are again tuples or sets. In the relational data model, however, the naturàl join is defined only for relations (that is, sets of tuples). We believe that it is important to treat every type of objects equally as operands for composition operations. Therefore, we need to generalize the operand type of natural join operator from a certain class of set-type objects (that is, sets of tuples) to more general types of objects including tuple-type objects and atmoic-type objects.

(b) Generalization of comparison operators
In relational databases, the natural join is based on testing equality of values of common attributes of two tuples, where each attribute value is a simple atomic value such as an integer and a character string. In the case of object-oriented databases, attribute values are not always those atomic values, but are general complex objects. For example, `[name:'Bob']` and `[name:'Bob', age:30]` are supposed to be tuple-type objects that are values of a

common attribute of two objects. By ordinary equality-testing, those two objects cannot be joined because the above two tuple-type objects are not equal. However, it is obvious that these two tuple-type objects have a kind of common information (in this case, name:'Bob'). Therefore, we need to generalize this equality-testing comparison operator so that it may handle such cases.

(c) Generalization into heterogenous set objects
Suppose that

{'Bob', 'John', [name:'Steve', age:20]}, and
{'Jane', 'Bob', [name:'Steve']}

are given set-type objects. In the spirit of the natural join, since 'John' and 'Jane' are not contained by both of them, these objects will not appear in the result. In this case, 'Bob' should appear in the resultant object. Also, from the discussion in (b),

[name:'Steve', age:20] and [name:'Steve']

are not equal, but share some common information.
So, some kind of object produced from these two tuple-type objects should also appear in the resultant object.

2.2 Behavioural Aspect of Objects

In CAD database situations, it is very frequent that users compose new objects from existing objects. Since objects encapusulate data (attribute values) and methods, it is necessary to consider which methods in existing objects are preserved after applying natural join operators. Figure 1 illustrates this problem. When new object o_{new} is made from two objects o_1 and o_2 as shown in Figure 1, the set of methods in o_{new} is not always equal to the union of the sets of methods in o_1 and o_2. Inituitively, if the method (name) m_1 is defined in both of o_1 and o_2, and the contents (*implementation*) of method m_1 in o_1 is not the same as the contents of method m_1 in o_2, then we can not decide the implementation of method m_1 for object o_{new}.

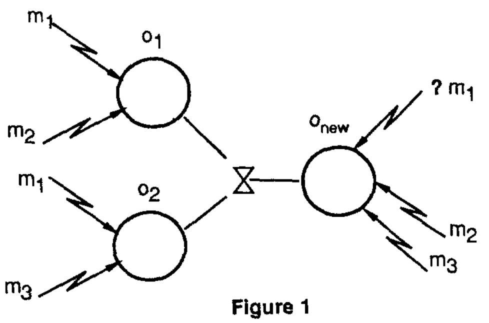

Figure 1

2.3 Basic Definitions

In this paper, we denote objects by $o, o_1, o_2, \cdots$, and method names by $m, m_1, m_2, \cdots$. For a given object o and an attribute a, $o.a$ denotes the value of the attribute a of o. For each object o, $attri(o)$ denotes the set of all the attribute names.

As for the definition of *objects* and the *equality*, we will obey the following definitions introduced by Bancilhon and Khoshafian[5]. In this sense, we will not consider the *object identity* in this paper.

[Definition 1] Objects
Objects are defined recursively as follows;

1. A number, a character, or a string is an object called an *atomic object.*
2. There are two special objects TOP ($\top$, the inconsistent object) and BOTTOM ($\perp$, the undefined object).
3. If $o_1, o_2, \cdots, o_n$ are objects, then the set $\{o_1, o_2, \cdots, o_n\}$ is an object called a *set object.*
4. If $o_1, o_2, \cdots, o_n$ are objects and $a_1, a_2, \cdots, a_n (n \geq 0)$ are distinct attribute names then, $o = [a_1 : o_1, a_2 : o_2, \cdots, a_n : o_n]$ is an object called a *tuple object.* o_i is denoted by $o.a_i$.

Then, we define our natural join in the following recursive manner.

[Definition 2] Natural Join
Given two objects o_1 and o_2, the *natural join* of o_1 and o_2, denoted by $o_1 \bowtie o_2$, is an object that is defined in the following recursive manner.

1. If o_1 and o_2 are the same atomic objects, then $o_1 \bowtie o_2 = o_1$.
2. If o_1 or o_2 is $\perp$ then $o_1 \bowtie o_2$ is $\perp$; otherwise if o_1 or o_2 is $\top$ then $o_1 \bowtie o_2$ is $\top$.
3. If o_1 and o_2 are tuple objects such that $o_1 = [a_1 : v_1, \cdots, a_l : v_l, b_1 : u_1, \cdots, b_k : u_k]$ and $o_2 = [b_1 : u'_1, \cdots, b_k : u'_k, c_1 : w_1, \cdots, c_m : w_m]$, (here a, b and c are attributes and u, v, w are attribute values (objects)) then $o_1 \bowtie o_2 = [a_1 : v_1, \cdots, a_l : v_l, b_1 : u_1 \bowtie u'_1, \cdots, b_k : u_k \bowtie u'_k, c_1 : w_1, \cdots, c_m : w_m]$. (Here, every $u_i \bowtie u'_i$ is neither $\top$ nor $\perp$. If at least one $u_i \bowtie u'_i$ is $\top$ or $\perp$, then $o_1 \bowtie o_2$ is defined to be $\perp$.)
4. If o_1 and o_2 are set objects such that $o_1 = \{e_1, \cdots, e_m\}$ and $o_2 = \{e'_1, \cdots, e'_n\}$, (where e and e' are the objects) then $o_1 \bowtie o_2 = \{e_i \bowtie e'_j \mid i \in \{1, \cdots, m\}, j \in \{1, \cdots, n\}$ and $e_i \bowtie e'_j$ is neither $\top$ nor $\perp\}$.

For example, Let o_1 and o_2 be the set objects such that $o_1 = \{1, 2, [a : 2, b : 3]\}$ and $o_2 = \{2, 3, [a : 2]\}$. By the definition of natural join, $o_1 \bowtie o_2 = \{2, [a : 2, b : 3] \bowtie [a : 2]\} = \{2, [a : 2, b : 3]\}$.

3 Realizing Natural Join

In this section, we show the implementation of our natural join operator in the Smalltalk-like message passing computing mechanism. That is, our natural join is also regarded as a method, and so, we denote the natural join operator by a message ⋈:, where the symbol ':' is used to represent that this message has an argument. Here, we assume to take a natural join of objects X and Y, which is denoted by X ⋈: Y. The implementations of the natural join are as follows:

(a) atomic class

```
| o |
self = Y ifTrue:[o ← self]
    ifFalse:[o ← 'undefined'].
↑ o
```

(b) set class

```
| o xi |
o ← Set new.                    "o = { }"
self do:[:x | Y do:[:y |        "x ∈ X, y ∈ Y"
    xi ← (x ⋈:y).
    (xi='undefined') ifFalse:[ o add:  xi]]].
↑ o
```

(c) tuple class

```
| o xi commonAttribute differentAttribute1
differentAttribute2 |
o ← Tuple new.
commonAttributes ← self attributres ∩:(Y
attributes).
commonAttributes do:[:c |
  xi ← ((self perform:a) ⋈:(Y perform:a)).
  xi= 'undefined'     "self perform:a = X.a"
    ifTrue:[↑ xi]
    ifFalse:[o add:a:xi]].
differentAttributes1 ← self attributes −:(Y
attributes).
differentAttributes1 do:[:b |
  o add:b:(self perform:b)].
differentAttributes2 ← Y attributes −:(self
attributes).
differentAttributes2 do:[:c |
  o add:c:(Y perform:c)].
↑ o
```

Here, o *add*:A denotes to add the object o to argument A.

Our natural join is defined in a recussive manner in which natural join operators are repeatedly propagated to each component object. So, this definition is naturally implemented in the message passing computing mechanism.

Let us show example joined the following objects o_1 and o_2 (that is, o_1 ⋈: o_2).
$o_1 = \{[name : x, dept : y, addr : [city : z]]\}$ and $o_2 = \{[dept : y, addr : [state : w]]\}$.
First, by ⋈: of a set object, this is $\{[name : x, dept : y, addr : [city : z]]$ ⋈: $[dept : y, addr : [state : w]]\}$ And, by ⋈: of tuple object, the above equation is $\{[name : x, dept : y$ ⋈: $y, addr : [city : z]$ ⋈: $[state : w]]\}$. Finally, by ⋈: of an atomic object and a tuple object, this result is $\{[name : x, dept : y, addr : [city : z, state : w]]\}$.
As for this definition, we should note the following.

- According to the types of operands of a natural join, necessary implementations of a natural join are invoked at execution time. If necessary, users can customize the implementations of natural joins according to the types of classes.
- It is not necessary to prepare a large intermediate space for exploring two objects for performing a natural join.

4 Method Preservation Under Natural Joins

In this section, we investigate what class of methods are preserved by the result of the natural join.

We denote the implementation of method name m defined in object o by $imp(o, m)$. In the implementation of method m_i, there often exists a statement which invokes another method m_j. This means that the method m_i can not be executed if the method m_j is not executable. Such dependency relationships among methods for a natural join $o_1 \bowtie o_2$ is represented by a multi-layered directed graph $G = (V, E)$, called *method dependency graph*, where V is a set of vertices where each vertex m corresponds to a method of o_i $(i = 1, 2)$ and their whole component objects, and E is a set of directed edges where each edge e corresponds to the dependency relationship between methods defined as follows.

- $\boxed{m_1} \longrightarrow \boxed{m_2}$ denotes that $imp(o_i, m_2)$ contains a statement `self m1`.
- $\boxed{m_1} \dashrightarrow \boxed{m_2}$ denotes that $imp(o_i, m_2)$ contains a statement `o m1`, where $o = self.a$ and $imp(o_i, m_2)$ contains `o m1`.

Suppose that method m is defined in both of o_1 and o_2 such that object o_1 and o_2 are to be joined. If the method m in o_1 is not same implementation as the method m in o_2, that is, $imp(o_1, m) \neq imp(o_2, m)$, then we say this situation as *method-name-conflict*. It is impossible for an object $o_1 \bowtie o_2$ to have methods with the same name and diffrent implementations.

Since our natural join is defined in a recursive manner, it is necessary to consider the method-name-conflicts at each nesting level.

Given a method dependency graph for object $o_1 \bowtie o_2$ the following procedure selects the methods preserved.

step 1 For each nesting level, delete the methods with method-name-conflict (Start from the deepest nesting level).

step 2 Delete all the methods that are reachable from the name-conflicting methods.

step 3 Go to the upper level of nesting, and repeat step 1 and step 2.

Here, we assume that each set object has that a fixed set of methods for usual set operations.
By above algorithm, we can obtain the methods which are preserved by the result of the *union* ($o_1 \cup o_2$). But in case of intersection, we add the following step to the above algorithm.

step 4 Delete all the methods of which implementation contain the losing attributes by intersection.

step 5 Delete all the methods that are reachable from the deleted methods by step 4.

[Example]
Suppose that the tuple object o_1 and o_2 with the attributes and methods (implementation) as follows.
$o_1 = [name : nishikawa, birthday : 1960.1.1, add : kobe, school : o_{11}]$,
$o_2 = [id : 001, birthday : 1960.1.1, add : kobe, school : o_{21}]$.
$o_{11} = [name : kobe, numberOfTeacher : 100]$
$o_{21} = [name : kobe, numberOfStudent : 1000]$
Here, we omit the implementations of methods which return the value of attribute as $\begin{pmatrix} whatsName \\ \uparrow name \end{pmatrix}$.
The method implementations of o_1 (To be omited the methods *whatsName*, *birthday*, *address* and *school*):

```
getAge
  | x1 x2 |
  x1 ← self thisYear.
  x2 ← self birthday.
  ↑ (x1-x2)
howOld
  | x1 |
  x1 ← self getAge.
  ↑ (x1+1)
changingSchool
  | x1 x2 |
  x1 ← self address.
  x2 ← self school whatKind
  (x1≠'kobe' or:[x2='small'])
  ifTrue:[↑ 'Change to Kobe university']
```

The method implementations of o_2 (To be omited the methods *whatsId*, *birthday*, *address* and *school*):

```
howOld
  | x1 x2 |
  x1 ← self thisYear.
  x2 ← self birthday.
  ↑ (x1-x2+1)
changingSchool
  | x1 |
  x1 ← self address.
  x1='kobe'
  ifTrue:[↑ 'Change to Kobe university']
```

The method implementations of o_{11} (To be omited the methods *schoolName* and *numberOfTeacher*):

```
whatKind
  | x1 |
  x1 ← self numberOfTeacher
  (x1>=100) ifTrue:[↑ 'big']
      ifFalse:[↑ 'small']
```

The method implementations of o_{21} (To be omited the methods *schoolName* and *numberOfStudent*):

```
whatKind
  | x1 |
  x1 ← self numberOfStudent
  x1>=999 ifTrue:[↑ 'big']
      ifFalse:[↑ 'small']
```

Here, in each implementation, *self* denotes a receiver object, and $o_i\ m_k$ denotes a resulting object obtained by applying (sending) a method (message) whose name is m_k to o_i. Each $| x_i |$ denotes a local variable x_i and $\uparrow$ returns the result of evaluating an associated expression.

The following points should be noted in the above implementations of methods:

- The method *howOld* and *changingSchool* are defined in both of object o_1 and o_2, respectively. But their implementations are all defferent. That is, they are in the states of method-name-conflict.
- In the implementation of method *changingSchool* in o_1, the method *whatKind* exists in it.
- The method *whatKind* is defined in both of object o_{11} and o_{21}. But their implementations are different. That is, they are in the states of method-name-conflict, too.
- The method *thisYear* is assumed to be inherited from each of their superclasses, and it is assumed not to be method-name-conflict.

We can obtain the method dependency graph of object $o_1 \bowtie o_2$ as Figure 2.
By the above algorithm, we delete the methods as follows:

1. Let's start from the deepest nesting level (o_{11} and o_{21}). Then delete *whatKind* because it is method-name-conflict.

2. Delete the method *changingSchool* defined in o_1 because *imp*(o_1,*changingSchool*) contains *selfschoolwhatKind*. That is, it is reachable from the name-conflicting method *whatKind*.

3. Let's go to a upper level of nesting. Then delete *howOld* because it is method-name-conflict.
Here, method *changingSchool* is method-name-conflict originally. But now this method-name-conflict disappears because *imp*(o_1, *changingSchool*) was deleted before step. So, method *changingSchool* in o_2 is preserved by the resulting object $o_1 \bowtie o_2$.

Therefore, we obtain the methods of $o_1 \bowtie o_2$ as follows; *whatsName*, *whatsId*, *thisYear*, *birthday*, *getAge*, *address* and *changingSchool*. Also, the set of preserved methods for the value of attribute *school* of $o_1 \bowtie o_2$ becomes
$\{schoolName, numberOfTeacher, numberOfStudent\}$.

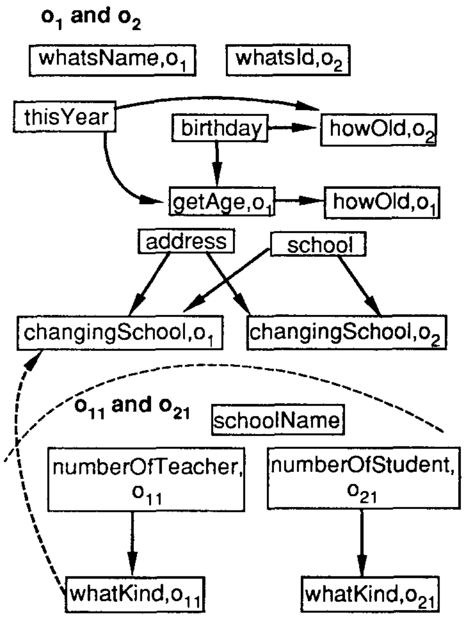

Figure 2

5 Properties of Object Composition Operations

The following are the definitions of *sub-object* relationship among objects, which is necessary to define *union* and *intersection* introduced by Bancilhon and Khoshafian.

[Definition 3] Sub-object relationship
Given an object o and an object o', we define that object o is a *sub-object* of object o' (denoted $o \leq o'$) recursively as follows;

1. Let o and o' be two atomic objects, then o is a sub-object of o' if $o = o'$.

2. Let o and o' be two set objects, o is a sub-object of o' if every element of o is a sub-object of an element of o'.

3. o is a sub-object of o.

4. Every object is a sub-object of $\top$ and $\bot$ is a sub-object of every object.

[Definition 4] Union and Intersection
The *union* of two objects o_1 and o_2 is the smallest object that contains both of them (i.e. their least upper bound under the sub-object relationship). It is denoted by $o_1 \cup o_2$.
The *intersection* of two objects o_1 and o_2 is the largest object which is contained in both of them (i.e. their greatest lower bound). It is denoted by $o_1 \cap o_2$.

Suppose that $o_1 = \{[a:1, b:1], [c:1]\}$ and $o_2 = \{[b:1], [a:1, c:1]\}$. We obtain the union, natural join and intersection of o_1 and o_2 as follows:

- $o_1 \cup o_2 = \{[a:1, b:1, c:1]\}$
- $o_1 \bowtie o_2 = \{[a:1, b:1], [a:1, b:1, c:1], [b:1, c:1], [a:1, c:1]\}$
- $o_1 \cap o_2 = \{[a:1], [b:1], [c:1]\}$

This example shows different aspects between our natural join, the union and intersection operators. As shown above, our natural join can be computed by computing each elemnt-wise natural join. On the other hand, the union needs a large intermediate space which holds the whole contents of o_1 and o_2. However, it should be also noted that the result of the union operator is guaranteed to be a *reduced object*, but that the result of natural join is not. Here, a reduced object is intuitively an object, in which there does not exist any nontrivial sub-object relationship between any pair of elements of the object.

In the following, we show several properties of object composition operations, some of which appeared in [6].

[Property 1]
If o_1 and o_2 are tuple objects such that $o_1 = [a_1 : v_1, \cdots, a_l : v_l, b_1 : u_1, \cdots, b_k : u_k]$ and $o_2 = [b_1 : u'_1, \cdots, b_k : u'_k, c_1 : w_1, \cdots, c_m : w_m]$,
then $o_1 \cap o_2 = [b_1 : u_1 \cap u'_1, \cdots, b_k : u_k \cap u'_k]$.

[Property 2]
If o_1 and o_2 are tuple objects such that $o_1 = [a_1 : v_1, \cdots, a_l : v_l, b_1 : u_1, \cdots, b_k : u_k]$ and $o_2 = [b_1 : u'_1, \cdots, b_k : u'_k, c_1 : w_1, \cdots, c_m : w_m]$,
then $o_1 \cup o_2 = [a_1 : v_1, \cdots, a_l : v_l, b_1 : u_1 \cup u'_1, \cdots, b_k : u_k \cup u'_k, c_1 : w_1, \cdots, c_m : w_m]$.

[Property 3]
Let o_1 and o_2 be set objects, then
$o_1 \cap o_2 \leq \{e_i \cap f_j \mid e_i \in o_1, f_j \in o_2\}$.

[Property 4]
For each object o_1 and o_2, $o_1 \cap o_2 \leq o_1 \bowtie o_2$ holds.

[Property 5]
For each tuple object o_1 and o_2, $o_1 \bowtie o_2 \leq o_1 \cup o_2$.

Next, we investigate the following question concerned with the relationship between natural joins and sub-object relationships.

- Are the sub-object relationships preserved under our natural join operation ?

[Property 6]
Suppose that o_1, o_2, o'_1 and o'_2 are atomic or set objects constructed by only set constructors. Then, if $o_1 \geq o'_1$ and $o_2 \geq o'_2$ then $o_1 \bowtie o_2 \geq o'_1 \bowtie o'_2$.

Note that when o_1 and o_2 are tuple objects, then *property 6* does not always hold in general [6].
The following condition guarantees that tuple objects satisfy this property.

[Property 7]
For each tuple object constructed by only tuple-constructors o_1, o'_1, o_2 and o'_2 such that $o_1 \geq o'_1$ and $o_2 \geq o'_2$, if the following conditions hold, then $o_1 \bowtie o_2 \geq o'_1 \bowtie o'_2$.

- $attri(o_1) - attri(o_2) \supseteq attri(o'_1) - attri(o'_2)$.
- $attri(o_2) - attri(o_1) \supseteq attri(o'_2) - attri(o'_1)$.
- $attri(o_1) \cap attri(o_2) \supseteq attri(o'_1) \cap attri(o'_2)$.
- For each attribute A in $attri(o'_1) \cap attri(o'_2)$, $x \geq y$ holds for the A-value x of $o_1 \bowtie o_2$ and the A-value y of $o'_1 \bowtie o'_2$

6 Concluding Remarks

In this paper, we introduced a recursively-defined *natural join* operation for composing CAD database objects. Then, we showed how to realize the natural join by the message passing computing mechanism. Next, we showed a way to examine what kind of behaviours (methods) are preserved under our natural join operations. Finally, we investigate mathematical properties about the relationships among several object composition operations (natural join, union and intersection).

Further research will be needed for the following problems:

- How to develop efficient processing techniques for object composition operations.
- Formalization and efficient implementations of other types of object composition operations, such as concatenation of objects, where the output of the former object is transmitted to the latter one.
- Obtaining a clear semantics of our natural join operation.

Acknowledgement

This work is partially supported by the Science Foundation Grant of the Ministry of Education, Science and Culture of Japan.

References

[1] Zaniolo,C. et al., *Object-oriented Database Systems and Knowledge Systems*, Expert Database Systems (L.Kerschberg ed.), Benjamin/Cummings Pub. Company Inc., pp.49-65, 1986.

[2] Bancilhon,F., *Object-Oriented Database Systems*, Proc. of ACM PODS, pp.152-162, March 1988.

[3] Tanaka,K., *Object-oriented Database Systems: Background and Concept* (in Japanese), bit, Vol.20, No.6, pp.83-90, June 1988.

[4] Batory,D.S. and Kim,W., *Modeling Conceptes for VLSI CAD Objects*, ACM TODS, Vol.10, No.3, pp.322-346, Sept. 1985.

[5] Bancilhon,F. and Khoshafian,S., *A Calculus for Complex Objects*, Proc. of ACM PODS, pp.53-59, March 1986.

[6] Tanaka,K. and Chang,T.S., *On Natural Joins in Object-Oriented Databases*, to appear in Proc. of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD89), Kyoto, Japan, Dec. 1989.