The Object-Oriented Relationship System for Managing Complex Relationships

Soochan Hwang and Sukho Lee

Department of Computer Engineering
Seoul National University
Seoul, 151-742, Korea

ABSTRACT

In the new applications such as OIS, CAD/CAM, and AI, it is required to support not only fixed Is-A and Part-Of relationships but also various user-defined relationships including complicate constraints. However, existing object-oriented systems have many weaknesses in managing those complex relationships. This paper presents the Object-Oriented Relationship system based on Object-Oriented Relationship Model (OORM) which is designed to provide facilities for representing and manipulating relationships modelled from the real-world. In the OORM, the relationship is expressed as a relationship object. It provides an abstraction mechanism for the association as a conceptual construct and enables to capture the semantics of the relationship more clearly such as constraints, generalization abstraction, and dynamic aspects. This paper also presents some techniques for implementing Object-Oriented Relationship system and operations for efficient manipulation of data using complex relationships.

1. INTRODUCTION

The object-oriented programming paradigm has been actively studied and has made much progress in various application systems such as office information systems (OIS) [Nier85, Guti89], CAD/CAM [Camm84], and AI [Shep84, Lee90]. The database area is also one of these fields [Zan86], and many object-oriented database systems have been developed, such as ORION [Bane87], Iris [Fish87], GemStone [Cope84], and others [Andr87, Lecl88]. These systems have enhanced the modelling capabilities of the database and have tried to complement the semantic scantiness in the traditional record-based database systems. However, these systems are lacking in representing and manipulating various complex relationships in the real-world.

In general, relationships between entities are classified into three categories: Is-A relationship, Part-Of relationship, and association [Kort86, Rumb87, Smit77]. So far, there have been many attempts to support Is-A and Part-Of relationships and much work has been done [Camm84, Kim87, Rumb87]. However, new applications such as OIS, CAD/CAM, and AI require to support not only such static and fixed Is-A and Part-Of relationships but also various associations defined dynamically. And they also require to represent and maintain various integrity and consistency constraints related to the associations. Even though existing object-oriented systems can represent and maintain all sorts of relationships through the reference between two objects, they have many limitations on modelling these relationships.

Another problem of object-oriented database systems pertaining to these relationships is that they do not provide the operation to navigate through complex relationships [Tsic88]. For manipulating very large databases containing many complex relationships among many classes, it is impossible to do with the operation for each individual instance only. Therefore, the object-oriented database system needs to have both of the facilities to navigate through multiple-level references among several objects and to get diverse information from several related objects.

The purpose of this paper is: 1) to describe how the semantics of various relationships in the real-world are represented and managed in object-oriented databases, 2) to present mechanisms for capturing and managing more knowledge such as semantic integrity and consistency constraints related to the object, 3) to show how relationships can be systematically managed and constructed as a generalization hierarchy, 4) and finally to show several implementation techniques for OORM and basic operations for efficient manipulation of database using relationships.

In OORM, we define the relationship object which provides an abstraction mechanism for the relationship as a conceptual construct and can capture the semantics of the relationship more clearly. We
have developed Object-Oriented Relationship (OOR) system to support concepts of OORM. It has been implemented in Smalltalk/ V [Digi87] and exploited several implementation techniques for supporting complex relationships.

The remainder of this paper is organized as follows. In section 2, we discuss some problems and limitations of the processing mechanism of the relationships in the existing object-oriented systems. Section 3 presents the basic features and the methods that can manage various dynamic relationships and their constraints in OORM. In section 4, we define the virtual class and generalization of relationships. Section 5 shows the implementation techniques and operations. Finally, a brief conclusion is presented in section 6.

2. PROBLEMS OF RELATIONSHIP HANDLING IN OODB AND RELATED WORKS

There are many kinds of relationships in the real-world. In general, relationships are distinguished by their mapping cardinalities such as one-to-one, one-to-many, and many-to-many, and also by the number of related entities such as binary relationships and n-ary relationships [Chen85, Ullm88]. There are also other relationships such as relationship among relationships [Kort86]. In the existing object-oriented systems, there are many problems and limitations in representing and manipulating the relationships in the real-world. In this section, we will discuss their problems and limitations with references to a sample ER diagram for a manufacturing company presented in figure 1.

![ER Diagram](image1)

Figure 1. An ER diagram for a manufacturing company

Firstly, existing object-oriented systems do not provide formal syntax and semantics to express relationships directly. Thus, the information about relationships may be stored redundantly and distributed to many participating objects rather than gathering it into a single object [Rumb87, Tsic88].

For example, representing a binary relationship as a pair of mutually interlocked attributes will lose semantic information, because the standard object-oriented model cannot represent the constraint that the two objects must point at each other [Rumb87, Tsic88, Ullm88]. But the recent models developed by [Andr87, Lee90, Mann89] provide the facilities to specify inverse-of constraint in the attribute specification. However, they can represent binary relationships only and cannot represent the attributes associated with the relationship itself.

Secondly, the new applications require the capability to specify integrity constraints not only over a single data item but also among many different data items associated with complex relationships. But in existing object-oriented systems, those constraints cannot be explicitly represented and managed as an element of the database system, so that the constraints have been partly expressed and its semantics have been buried in the method of a class even though they are related to many different classes [Hwan90]. For example, consider an integrity constraint that Supplier, Part, and Project in the association PSJ, must be all colocated. The constraint among those classes can only be expressed separately in each class as a method and it is very difficult to enforce the constraint.

Thirdly, since all relationships are represented as a reference between two objects in the object-oriented paradigm, some relationships, such as n-ary relationships and relationships among relationships, cannot be properly represented in the object-oriented database. Thus, those relationships associating objects of n classes are merely represented by n*(n-1) decomposed binary relationships. In functional object-oriented approaches [Fish87, Mann89] where all operations are expressed by functions, they need to define 2^n related functions to manipulate the n-ary relationship. For example, in the object-oriented systems, the ternary relationship PSJ in figure 1 can be replaced by three binary relationships: Part-Supplier, Supplier-Project, and Project-Part. However, if we want to get some information from these decomposed binary relationships, we will get some nonfacts which do not exist in the original relationship [Chen85].

Let us consider another relationship describing information about employees who work for a particular project and use many different machines for that project. The Use relationship is a relationship among relationships. In order to represent such relationships, an abstraction mechanism to treat the relationship Work as a higher-level entity is needed. However, the existing OO systems do not support such an abstraction.

There are approaches [Cope84, Roll89, Rumb87] to represent the
relationship as a separate aggregate object. In [Cope84,Roll89], a relationship can be represented as a first class object using the aggregation concept. In these models, a class representing a relationship consists of attributes which refer to the related objects and the relationship attributes. However those classes representing relationships cannot properly express the semantics of the relationship such as cardinality, referential transparency between aggregate object and participating objects, and various constraints. Users have to maintain all the semantics. In [Rumb87], a relationship and its cardinality are represented as a set object called relation whose elements are composed of the oids of participating objects. But an object cannot reference related objects or relations directly, because their elements are not first class objects. And in the above models, since the aggregate object contains only oids of participating objects, operations are only applied to those participating objects as the target of a method.

3. BASIC FEATURES OF OORM

In this section, we briefly describe the basic concepts of OORM: objects, properties, constraints, concatenated objects, and relationship objects. Each concept is explained with references to a sample schema definition of figure 2. Figure 2 shows the class definitions of entities and relationships of figure 1. More details about OORM can be found in [HwaL90,Hwan90].

Class Employee super Person
( salary : integer with [x] x between:10000 and:1000000],
friends : { Person }, affiliate : Department )

Class Project super Rclass
( pName : string, city : string, manager : Employee )

Class Work for Employee(*), Project(*) super Rclass
( effort : integer with effortConst )
method effortConst
(Work where: [:wl w Employee = (self Employee)])
sum: effort < 100.

Class PSI for Project(*), Supplier(*), Part(*) super Rclass
constraint Colocation
[ (self Part city = (self Supplier city)) and:
self Supplier city = (self Project city)]
( qty : integer )

Class Use for Work(*), Machinery(*) super Rclass
( qty : integer )

Figure 2. Definition of classes

3.1 Objects and Classes

In OORM, all entities and concepts are represented as objects that

are uniquely identified with their own object identifiers (oids). The objects belonging to a class have common properties. The OORM distinguishes the type class used as a domain of an attribute from the class representing an entity or a relationship.

Type class provides a facility to define an abstract data type that enables a user to offer flexible domains of attributes. OORM defines basic type classes such as Integer, Char, String, and Boolean. Users can also define their own data types. In this paper, we will simply call the type class type.

The class of OORM represents an entity or a relationship. All classes of OORM participate in the class hierarchy according to their Is-A relationships between classes. The root of the class hierarchy is a system-defined class called Object and all user-defined classes become a subclass of the class Rclass defined by system. In order to support associative queries, OORM implicitly defines a set object for each class which is a set of all instances of a class. We call the instance object an instance set. The instance set corresponds to the relation extension in the relational model.

3.2 Attributes and Methods

The properties of a class consist of attributes and methods. The attributes hold the state of an object. An attribute can be specified by the domain and its constraint. The domain of an attribute can be either a type or a class. The attribute value of a type is a real value and the attribute value of a class is an oid of related object. The attribute defined on a class represents the relationship with other class referenced through the attribute. We call this attribute reference attribute. The set-valued attribute is defined by specifying the domain within braces, { }, such as attribute friends of class Employee.

A constraint can be specified by with-clause in the attribute definition. This attribute constraint must be satisfied by the attribute values. The constraint can be represented as either a block of codes whose result is a Boolean value or a name of Boolean method which is defined in the method definition part. For example, the Boolean method effortCons of figure 2 is used as the attribute constraint that the total effort of an employee cannot exceed 100%. The constraint on the attribute salary is expressed in the attribute specification with the block of codes. In the block, the variable x which is similar to a block argument of Smalltalk/ V [Digi87] represents the attribute salary.

The method specifies a behavior of an object. It is composed of a method name, parameters, and a method body. For each attribute, OORM defines two methods, called attribute methods,
for attribute retrieval and update. In OORM, a set of operations is defined to manipulate OO database, and those operations are implemented as an extension of the Smalltalk/V primitive methods which are used for specifying the method. It will be described in section 5.4.

3.3 Concatenated Object

One of new features of OORM is that several kinds of objects can be grouped into a single object and share their properties without losing the advantages of data encapsulation. This operation is called object concatenation. Suppose objects *anEmployee* and *aProject* are the instances of class *Employee* and class *Project* respectively. The object concatenation of these two objects, expressed as *anEmployee*aProject, logically includes all the properties of both objects. Thus, all messages related to the two objects can be sent to this concatenated object without specifying an exact receiver. Actually, the concatenated object is not a physically combined object but an ordered collection of oids of those two objects. We will explain the implementation details about the concatenated object in section 5.3.

On the basis of the object concatenation, cartesian product can be defined. It combines information from several classes, and provides the means that operations can be applied to all objects of several classes at a time. The formal definition of cartesian product of two classes *S* and *R* is as follows:

\[ S \times R = \{ s \times r \mid s \in S \text{ and } r \in R \} \]

3.4 Relationship Object

OORM can define the relationship object which represents a relationship. The relationship object is a logical entity which contains related information and constraints of a relationship and belongs to a relationship class. Following is the definition of relationship class *R* where *R* is an association among classes *E1*, *E2*, .., *En*.

\[ R = (E1 \times E2 \times \ldots \times En, C, P) \]

- **C**: constraints on relationship *R*
- **Ei**: participant classes of relationship *R*
- **P**: set of relationship properties

A relationship class represents an association among several classes. Formally, it is a mathematical relation on \( n > 1 \) (possibly nondistinct) classes. If *E1*, *E2*, .., *En* are classes, then a relationship class *R* is represented by a cartesian product of a list of participating classes and the properties of the relationship itself *P*. And the relationship is constrained by the constraint *C* restricting the cartesian product. The properties of *R* include not only the properties of the relationship itself but also all the properties of the participating classes using the concept of the object concatenation. Thus, all the information related to a relationship are represented in the relationship class, and the operations to the relationship are uniformly applied to all participant classes. More details about the sharing of properties are described in section 5.1.

In OORM, all classes are handled uniformly. Thus, a class *E* representing an entity is considered as a special kind of the relationship class which has an empty set of participant classes. That is, *E* = (*C*, *P*), where *C* and *P* are the constraints and the properties of the class *E*, respectively. The constraint mechanism of OORM enables users to add constraints in parallel with schema and data generation and to provide the means of incorporating knowledge into the database. For example, in CAD/CAM applications, various standards in the design, engineering, and manufacturing process can be expressed as constraints.

Then, let's consider the class specification in OORM. In the class definitions of figure 2, the for-clause specifies participating classes with their cardinalities, where the defined class represents a relationship. The participants are referred by their class names in the method. The cardinality of a participating class may be specified with one(1) or many(*). The constraints-clause specifies the class constraint that must be satisfied by all objects of the class.

For example, the class *Work* represents the many-to-many relationship between *Employee* and *Project* including the relationship attribute *effort* and its attribute constraint *effortConst*. The binary relationship between *Machinery* and *Work* is also represented as the relationship class *Use*. In the specification, the relationship class *Work* is treated as an aggregate object which is an abstraction of *Employee*, *Project*, and its relationship *Work*. The ternary relationship among *Project*, *Supplier*, and *Part* and its integrity constraint that forces all participants to be colocated, are represented as the relationship class *PSJ*. The constraint *Colocation* is enforced when the relationship is created and modified, and the relationship object which does not satisfy the constraint cannot exist.

In relationships, there is a relationship between the same entity. Let's take an example relationship, *Manufacture*, that a part consists of other parts. Since a part object can participate in many relationships in different roles, OORM allows one to specify the role name (alias). The different roles that an object may play in a relationship are specified in the definition of the participant class. *Manufacture* relationship can be defined as follows:
Class Manufacture
for (Part(*) Component), (Part(*) Complex)
super Rclass
constraint Size
[ self Component size < (self Complex size) ].

3.5 Semantics of Relationship Object

The relationship object is an instance of a relationship class and has similar characteristics to complex object in the sense that a set of related objects is treated as a single logical entity. A complex object consists of exclusive component objects and represents the Part-Of relationship between objects. The component object is one whose existence depends on the existence of the complex object and is owned by exactly one complex object [Bane87]. On the other hand, a relationship object consists of nonexclusive participant objects and represents an association relationship between them. Its existence depends on the existence of each participant object. Therefore, if either any one of participant objects within the association or the association itself is deleted, the relationship object representing the association is also deleted.

Then, let's compare the relationship object with aggregate object of [Cope84,Roll89]. In both approaches, relationships can be represented as separate first class objects.

In our approach, relationship object includes both the relationship’s attribute and participant information as a form of concatenation. Thus, in this object, we can get all information related to the relationship without user’s explicit navigation, because object concatenation provides the facility to share properties.

However, even though aggregate object can also include the relationship’s attribute and participant information, it can only contain participant information as attributes referencing participants. Thus, information about a relationship is distributed to many objects and it needs user’s explicit navigation from the aggregate object to participants. And it is very difficult to express the semantics of relationships such as cardinality and constraints.

4. VIRTUAL CLASS AND GENERALIZATION OF RELATIONSHIPS

OORM allows one to build new classes on the basis of existing ones, and provides alternative interfaces to the instances of existing classes. This scheme is called virtual class. The main advantage of the virtual class is to provide users with multiple views of data [Lee90,Tana88]. The notion of virtual class and schema virtualization is proposed by [Tana88]. However, the virtual class of [Tana88] cannot exist as a super/subclass of the real class in the class hierarchy. The virtual schema can only be defined as an alternative to the real schema. This restriction reduces the utilization of the virtual class.

In OORM, the virtual class is defined by specifying derivation constraint on existing base classes and its instances can be derived from the instances of the base classes. In the class hierarchy, a virtual class can be located as a subclass of real class or another virtual class. It can also have real classes as its subclasses and inherits the derivation constraint to them as a class constraint.

In addition, the virtual class of OORM provides the means to define and manage complex relationships dynamically. That is, virtual relationships can be represented in the database. The virtual relationship class can be defined in two ways: Firstly, it can be directly defined by participating classes. Secondly, it can be derived from the real relationship classes as a subclass of them.

Let’s consider the Potentially-Supply relationship that describes the list of potential suppliers for a given part and is a many-to-many mapping. This relationship can be dynamically represented as a virtual relationship class using the first method. The definition of the virtual class is as follows. It is assumed that a supplier potentially supplies those parts in the same city that the supplier is located in.

Vclass PotentialSupp for Part(*), Supplier(*)
super Rclass
where [ self Part city = (self Supplier city) ].

The syntax of a virtual class specification is similar to that of the base class except for the message receiver. The where-clause specifies the derivation constraint which represents the method to derive instances from the base class and at the same time represents the class constraint of the virtual class. The derivation constraint is expressed in the same way of the class constraint of the real class.

When a virtual relationship class is defined as a subclass of real class, Is-A relationship is formed between the base relationship class and virtual one. In OORM, relationships can also construct a generalization hierarchy according to their Is-A relationships like the general classes. A specialized relationship of a relationship can be defined by:

1) Modifying the cardinality of a relationship, but the only change is one of reduction.
2) Adding constraints to a relationship. Only the added constraints
are specified and connected to the inherited constraints by and-condition.

3) Adding attributes and methods like the general classes.

Let's take an example of figure 3 representing a generalization hierarchy of the virtual class and Is-A relationships between relationships. For example, the relationship Work of figure 3 can be specialized into two subclasses such as MajorWork, where the effort of an employee is greater than 50% on a project, and MinorWork, where it is less than 50%. Then the cardinality of MajorWork is modified into many-to-one between Employee and Project, because the total effort of an employee cannot exceed 100%.

These two subclasses can be defined as virtual relationship classes with the above constraints on effort. If the manager of a project must be a major worker in the project, the relationship Manager can be defined as a subclass of the relationship MajorWork. Figure 3 shows the Is-A relationships between relationships and its definitions in OORM.

![Diagram of relationships](image)

Class Work for Employee(*), Project(*) super Rclass

- `effort : integer with effortConst`

Vclass MinorWork super Work

- `where [ self effort <= 50 ]`

Class Manager super MajorWork

5. IMPLEMENTATION OF OORM

In this section, we describe the major considerations for the implementation of OOR system. The system has been implemented in Smalltalk/V by adding classes and properties supporting the concept of OORM. The figure 4 shows the major classes in the hierarchy of OOR system.

![Diagram of basic class hierarchy](image)

In the class hierarchy, the class Type is a superclass of all type classes which are defined by the system and provides the domain of attributes. The class Rclass is a superclass of all the classes which represent entities or relationships. It provides the operations and properties related to the management of entity and relationship classes. And the instance set of each class exists as an instance of the set class RclassSet in which all set operations for the instance set is implemented. The instance set is defined when the class is defined and referred by a global variable whose name consists of the class name with suffix _set. For example, the instance set of the class Employee is referred by the global variable EmployeeIset.

5.1 Class

Firstly we describe the implementation of the entity and relationship class. In Smalltalk/V each class object contains all schema information about instance objects and is also an instance of the class Class. The major information of the class object is as follows.

- `superClass : contains the superclass of this class.`
- `messageDictionary : contains a dictionary of all the methods defined by this class. The key is the method selector and the value is the compiled code.`
- `subclasses : contains an array of all the subclasses of this class.`
- `instances : contains an array of instance variable names defined by this class. The names are stored as strings.`

OOR system adds instance variables to the original class object of Smalltalk/V for representing the definition of attributes, constraints, and the information about the participating classes of a relationship. The added instance variables and their functions are as follows.

- `attributes : contains a dictionary of attribute specifications because the original Smalltalk/V does not support typed attributes. The key is an attribute name and the value is an array consisting of attribute type and constraints.`
0 pclasses : contains an array of participants. Each element represents an information about a participating class such as the name of participating class, cardinality, and its alias(role name). If a subclass is defined without specifying participants, this variable is copied into that of the subclass.

0 constraints : contains an ordered collection of class constraints. A class constraint is represented by the constraint name and the constraint block stored as a compiled code.

0 prelations : contains an array of relationships in which this class participate. It contains the relationship class and its alias for supporting referential transparency. This information is used for deleting relationship objects when an instance object of this class is deleted.

For each class, there are attributes and methods which are automatically added to the user-defined properties. To every user-defined class, a system-defined attribute pobjects is added and its value is defined as a concatenation of all participant objects when this object is a relationship object. For example, if the relationship object w of Work represents the relationship that an employee e works for the project p, then w includes the information of participants in the attribute pobjects as a form of e[p. For processing the message to the relationship object, the relationship object itself is concatenated to the attribute and the message is forwarded to this concatenated object. When the attribute message pName is sent to w, it is forwarded to the concatenated object wll(e[p) and the name of the project p is returned. The value of the attribute pobjects is Nil for an entity object.

For each user-defined attribute in a class, the system defines two attribute methods with the same name as the attribute. The one is for retrieving attribute and the other is for updating. The retrieval method simply returns the value of the attribute. The update method updates the attribute value and checks both domain and attribute constraints. And the class constraints are also checked after updating attributes. When one of these constraints is violated, the update message is cancelled and it returns an error message. The value of an attribute can be either the object of declared domain class or its subclasses. The example of figure 5 shows two system-defined attribute methods for the attribute salary of the class Employee. In figure 5, the method checkConstraints is a system-defined Boolean method that applies all class constraints specified in the class and its superclasses. The method returns false if this object violates any one of class constraints.

salary

"attribute method for salary"

"salary:

salary: aValue

"update attribute method for salary"

<table>
<thead>
<tr>
<th>const</th>
<th>save</th>
</tr>
</thead>
<tbody>
<tr>
<td>save := salary.</td>
<td></td>
</tr>
</tbody>
</table>
| "domain check"
| (aValue class = Integer or Integer allsubclasses includes:(aValue))) |
| ifTrue: |
| "attribute constraint" |
| const := [ x | x between:10000 and:100000]. |
| (const value: aValue ) |
| ifTrue: [salary := aValue] |
| ifFalse: [self error:'attribute constraint violation'] |
| ifFalse: [self error:'attribute type must be Integer']. |
| "class constraint" |
| (self checkConstraints) |
| ifFalse: [salary := save. |
| self error:'class constraint violation']. |

Figure 5. Example definition of attribute method salary

5.2 Virtual class

The virtual class is defined and managed with the same way as real classes but it is an instance of the class Vclass. The derivation constraint is also treated in a similar way to the class constraints. The derivation constraint of the virtual class is stored in the instance variable constraints as an element with the system-defined name iSetConstraint. Using this constraint, OOR system defines the method makeIset that creates an instance set of a virtual class when a message is sent to. The figure 6 shows the instance method makeIset defined in the virtual class MajorWork. The instance set MajorWorkIset, which is created by the method, is a set of oids of Work instance objects satisfying the derivation constraint.

makeIset

"make instance set of Virtual Class"

| deriveRule |
| (((self class) constraints) do: [ x | |
| ((x at: 1) = #iSetConstraint) |
| ifTrue: [ deriveRule := x at: 2 ] ]. |
| Workset do: [ x |
| (deriveRule value: x) |
| ifTrue: [ MajorWorkIset add: x ] ]. |

Figure 6. Example definition of the method makeIset for the virtual class MajorWork

Next, we will explain the processing mechanism of the messages to the virtual classes. Since the instance of a virtual class is really a reference to the instance of the real class, the messages whose corresponding methods are defined in the virtual class cannot be
processed. Let's assume we want to define an original method overeffort of the virtual class MajorWork, which returns the value (effort - 50). If the method is not defined in real classes, on receiving this method the instance of the virtual class will return error message in ordinary case. In this case there needs a mechanism to process the message using the method of the virtual class. For this mechanism, there is an approach [Tana88]. In the implementation of [Tana88], first the real class to which the instance object directly belongs is searched. And the search is repeated upward to the root of the base schema. Finally, if the root of the hierarchy does not have the method overeffort then continues the search of the starting from the virtual classes. In [Tana88], there is an explicit distinction between real class(schema) and virtual class(schema).

However, there is no explicit distinction between the real class and the virtual class in OORM. The virtual class may exist as a superclass or subclass of the real class. Thus, when a message is sent to the virtual instance, the receiver of the message is treated as an instance of the virtual class and applies the method of the virtual class according to the user's perspective. And if the class does not have the method, the search begins upward to the superclass whether it is a real class or not.

A real class can also exist as a subclass of a virtual class. In that case also, the real class inherits all properties from the virtual superclass. For example, if the message overeffort is sent to the instance object of the class Manager, it is processed by the method of the class MajorWork and returns the result. The derivation constraint of the superclass is also inherited to the class as a constraint.

5.3 Concatenated Object

Concatenated object supports the grouping and property sharing of different objects. The sharing of properties can be implemented by a message protocol between objects. If a message m is sent to the concatenated object s|r of two instances of classes S and R, the message m is processed as follows:

\[(s|r) m = (s \text{ if } m \in Ps) \text{ or } (r \text{ if } m \in Pr \text{ and } m \in Ps)\]

where Ps and Pr are sets of properties of objects s and r, respectively

In the above protocol, if method m is existed in both classes, the method of object s has the precedence because the order of searching properties depends on the order of concatenation. Thus, in order to specify a receiver class, the class in which the message has to be processed should be specified by a qualified name like S@m. It modifies the searching order of properties in concatenated object.

We define the class Concat as a subclass of the class Ordered Collection, which is a class of concatenated object. The messages to the concatenated object is processed as follows; First, for processing the messages to the concatenated object itself, the message is applied to the concatenated object and if it returns error, then the message is sent to the first component object. And if the result is an error, it is sent to the second component, and so on.

The brief algorithm to implement the message protocol is as follows.

```plaintext
xmsg: aMsg
    answer := self aMsg.
    isError
    ifTrue:
        for each component object do: [:component |
            answer := component aMsg.
            isError
            ifFalse: [ *answer ] ]
    ifFalse: [ *answer ].
```

The method xmsg is defined in the method of the class Concat and processes the messages to the concatenated object.

5.4 Methods for Object Manipulation

In OOR system, various methods are defined to provide facilities for manipulation of the objects. Those methods are implemented by the primitive methods of Smalltalk/V and provide the useful operations to the object-oriented database system.

**Basic Set Operations**

OOR system provides the conventional set operations such as set union(+), intersection(*), and difference(-). Those operations have their usual meanings. Given two sets belonging to the same (sub)class of Set, those operations create a new set object in that class.

**Object Selection**

The object selection operation is used to extract qualified objects from the instance set of specified class. OOR system defines two methods for this operation, where: and find:. The where: message returns a set of all objects that satisfy given conditions, and the find: message returns the first object satisfying the given conditions.

In the following examples, (a) returns a set of employees whose salaries are more than 30,000 and (b) returns a single employee object whose name is 'Lee'.

(a) Employee where: [:e | e salary > 30000 ]

(b) Employee find: [:e | e name = 'Lee' ]
In the above operations, a Boolean block has to be specified as a parameter representing qualification. The qualification is expressed by various operators: relational operators, logical operators such as and:, or:, not, and usual quantifiers for set-valued attributes such as exist: and all:. The quantifier messages also have a Boolean block as a parameter and return a Boolean value.

**Cartesian Product**

Cartesian product builds a set of all possible concatenated objects which consist of the object from each of the two specified classes. The method `product:` is provided as an implementation of this operation. The following `product:` message returns a set of all possible concatenated objects of employees and projects.

```
Employee product: Project
```

In addition, we provide the message `isComponent:` that can be used to test the inclusion of a concatenated object in the other concatenated object. That is, given two concatenated objects `p` (not necessarily concatenated object) and `c`, `c isComponent: p` is true, if all component objects of `p` exist in `c` with the same order of appearing in `p`. But they need not consecutively exist in `c`. Followings are examples of the message.

```
o1|b2|b3 isComponent: a2 = True
o1|b2|b3 isComponent: a1|a3 = True
o1|b2|b3 isComponent: a3|b1 = False
```

**Referential Join**

The purpose of referential join is to combine referencing objects and referenced objects, so that operations can be applied to all objects of classes related to an association. If there is a reference from class `S` to class `R` through a reference attribute `a` of `S`, the referential join of `S` and `R` is the set of all concatenated objects that are composed of the referencing object of `S` and the referenced object of `R`. The referential join is implemented by the method `join:`. The following example of figure 7 shows the referential join between two classes `Employee` and `Department` through the reference attribute `affiliate` where an arrow represents a reference relationship.

```
Employee -- affiliate --> Department
  e1     d1
  e2     d2
  e3

Employee join: affiliate = { e1|d1, e2|d2, e3|d2 }
```

**Projection**

In OOR system, attribute methods are provided for retrieving attribute values from an object. To retrieve all attributes from a class or a set of objects, we provide the method `project:` whose argument is an array of attribute names. For example, to get project names and efforts for projects for which employee 'Kim' works, the message is expressed as follows. The result of the message is a set of an array composed of project name and effort.

```
(Work where: [ w | w name = 'Kim' ]) project: (pName effort)
```

In the above example, when the message is sent to a `work` object, the `work` object is concatenated to the participant objects which are also stored as components of the concatenated object in the system-defined attribute `pobjects`. Thus, we can apply the messages `name of Employee` and `pName of Project` to `work` object.

The method `project:` can be also used to extract some component objects from a set of concatenated objects and it returns a set of concatenated objects which are components of the concatenated object. The message for that operation is simply expressed by the list of the names of component classes to be retrieved. Following example shows the projection of supplier objects supplying some parts to the project 'Sorter' from the set of work objects.

```
(Work where: [ w | w pName = 'Sorter' ]) project: (Supplier)
```

**6. CONCLUSION**

We have presented in this paper a data model OORM and its implementation methodology. In OOR system, the relationship is expressed as the relationship object. It can provide an abstraction mechanism for the association as a conceptual construct and capture the semantics of the relationship and its constraints more clearly. The main contributions of OOR system are as follows:

1) Relationships are treated as higher-level objects through the abstraction concept of the relationship object, thus the information to be related to the relationship can be treated and processed as a unit.

2) Since a relationship represents an inherent constraint between objects of two or more classes, it can be specified abstractly without imposing an implementation.

3) Semantic integrity and other consistency constraints can be explicitly represented and maintained as related information changes.
4) Both navigation and set operations are provided and operations can be uniformly applied to all related objects as a whole.

5) Generalization of relationships and virtual relationships are supported in order that the complex relationships in the real-world can be systematically modelled into the database.

7. REFERENCES


