Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 76 (2005) 171–180
The Journal of Systems and Software
www.elsevier.com/locate/jss

Design and implementation of an extended relationship semantics in an ODMG-compliant OODBMS

Hyun-Joo Lee [a], Sang-Won Lee [b,*,1], Hyoung-Joo Kim [c,2]

[a] Department of Computer Science, University of Southern California, Los Angeles, CA 90089, USA
[b] School of Information and Communication Engineering, Sungkyunkwan University, Chunchun 300, Jangan, Suwon, Kyounggi 440746, South Korea
[c] Computer Science and Engineering, Seoul National University, San 56-1, Shilim, Gwanak, Seoul, South Korea

Received 22 July 2002; received in revised form 20 April 2004; accepted 26 April 2004
Available online 7 June 2004

Abstract

Relationships, in addition to entities, are important in real-world database modeling. In particular, many object oriented database applications including CAD/CAM, CASE and multi-media need to model various and complex relationships, especially the 'part–whole' relationship. Without the built-in relationship supports from DBMSs, there is a huge overhead in managing relationships from application development to maintenance, since the relationships should be hard-coded within the application program itself.

In this paper, we propose a powerful 'part–whole' relationship model, which naturally extends the ODMG-3.0 object database standard. The proposed relationship model can support almost all of the relationship functionalities existing in the contemporary relational database model and the object oriented data model. In order to design and implement this relationship model, we seamlessly extend the ODMG-3.0 relationship using the inheritance concept. Also, we identify several possible run-time anomalies in implementing the relationship and provide solutions for their problems.

Keywords: Relationships; Referential integrity; ODMG-3.0; Object databases

1. Introduction

The relational data model (Codd, 1970) is very good for modeling data with simple structure. Basic data items are represented with rather short and fixed-length records and these records constitute a table. The relationship among entities can be represented with the concept of primary and foreign key. However, this relationship mechanism in the relational model has the limitations in the application areas where various and complex relationships are needed.

On the other hand, the object-oriented data model is powerful enough to represent a complex object as a recursively nested object (Kim, 1987). The object-oriented data model represents a real world entity as a single unit object, which has both structural and behavior properties. An object models a real world entity and the state of the object is only altered by methods of the object itself. A class is a collection of objects that model the same kind of objects and there is a hierarchical relationship between classes.

Although the modeling power of the object-oriented model is powerful, it has still limitation in representing a collection of objects as a single *logical* unit. Namely, it lacks the 'part–whole' relationship support (Bertino, 1998; Halper et al., 1994, 1998). Many application areas such as CAD (computer-aided design), CASE (computer-aided system engineering), and information repositories need to define and manipulate several related objects as a single logical unit (Lochovsky, 1985;

* Corresponding author. Tel.: +82-31-290-7971; fax: +82-31-290-7211.
E-mail addresses: hyunjul@usc.edu (H.-J. Lee), wonlee@ece.skku.ac.kr (S.-W. Lee), hjk@oopsla.snu.ac.kr (H.-J. Kim).
[1] Supported in part by grant no. R05-2003-000-11943-0 from Korea Science and Engineering Foundation.
[2] Supported in part by the Brain Korea 21 project.

0164-1212/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2004.04.023

Kim, 1987). Bernstein (1998) identified some relationship management functionalities in information repositories and addressed the shortcomings in relationship management of object-oriented database management systems (OODBMS). Also, the relationship management is one of important issues in open hypermedia systems to resolve dangling problem in broken hypertext links (Davis, 1998). Semantic relationship management is one of methods to address this problem. In particular, XLink managements in XML (W3C, 2001a,b) repositories require this property.

In recent years, the importance of relationship management has been underscored and many other groups have studied in this issue (Albano et al., 1991; Jagadish, 1992; Kim, 1987). From these works, we can summarize the necessities of the built-in support for relationship in DBMS as follows. First, it is possible to explicitly represent the various relationships in a schema. Therefore, users can easily understand the logical structure of objects and the interdependency among objects such as the effects of propagations of an operation on an object to others. Second, because of simple representation of complex operation of objects through the schema, errors in developing the application programs also can be dramatically reduced. Finally, after deployment of applications, it is much easier to maintain codes when the change of semantic of an inter-object relationship is required. Without the built-in relationship support, all related application logics should be located and changed. In contrast, with the built-in support, only the schema need to be changed and the applications are just recompiled. As a minor side benefit, the program code size can be considerably reduced because the application logic, without explicit relationship support from DBMS, should hard-code the relationship management modules wherever necessary.

The object model in ODMG-3.0 (Cattell et al., 2000), the standard of object-oriented databases, has two kinds of properties to represent the object state. One is the *attribute* that defines the state of a type. The other is the *relationship* that relates objects of two types and each type must have instances that can be referenced by object identifiers. A relationship specification names and defines a traversal path for the relationship. In C++ binding of ODMG-3.0, this traversal path is represented by template class d_Rel_Ref <class T, const char* M>. A class, which has this template class as a variable, refers to the template class T and the template class T also refers to the class. The variable name in the template class T, which refers to the class, is M. The representation of relationship in ODMG-3.0 C++ binding guarantees the referential integrity, that is, when a referenced object is deleted, the pointer in referencing side is automatically reset to null. The problem of the relationship in ODMG-3.0 is to define the relationship as the static part of each object. It provides only the static referential integrity but does not support the behavioral semantics among relationships which are essential in 'part–whole' relationship.

In this paper, we propose a systematic behavioral semantics for a 'part–whole' relationship that seamlessly extends the static relationship of ODMG-3.0 and add methods to represent these relationships. Our contributions are twofolds: (1) we propose a model of relationship semantics based on *the* '*part–whole*' semantics and represent this model by extending C++ binding of ODMG-3.0, (2) we implement the model in a repository system Soprano (Ahn et al., 1996) of SOP (SNU OODBMS Platform), ODMG-3.0 compliant OODBMS.

The remainder of this paper is organized as follows. In Section 2, several relationship semantics in object-oriented modeling fields or OODBMS are discussed. In Section 3, we conceptually explain our extended relationship model. In Section 4, we describe how to bind this extended relationship model into ODMG-3.0 C++ binding environment and provide a usage example. In Section 5, a few subtle implementation issues are discussed and our solutions are given. Section 6 concludes the paper.

2. Related works

There are two kinds of related works. First, several works systematically defines specific semantic relationship such as a parent–child relationship and a collection. Second, other works focus on the referential integrity between two objects.

Examples of the first case are relationships in UML (Rumbaugh, 1987; Rumbaugh et al., 1999) and relationships (Peckham et al., 1995) in data modeling system SORAC. In UML, relationships among objects can be categorized as *generalization*, *association*, and *aggregation*. The generalization means IS-A relationship. If an object O1 has IS-A relationship with an object O2, all members of the object O2 also become the members of the objects O1. Usually the concept of generalization is directly supported by most object-oriented language and database systems. Association is also directly supported by UML and most systems. However, the concept of aggregation in UML is not sufficiently supported. An aggregation represents that an object is composed of several part objects, which is the part–whole relationship. For example, a car is composed of tires, an engine, and doors. The aggregation can degrade the complexity of schema design by treating several objects as a single unit. In UML, the aggregation only defines the abstract semantic, not behaviors. On the other hand, our relationship model can provide simple and flexible ways for modeling associations.

Jagadish treats the relationship as a vehicle to maintain integrity, and suggests the functionalities to support

the integrity in OODMBS (Kim, 1987). These functionalities include relational integrity, referential integrity, and uniqueness, which represent other objects' behaviors depending on an object in its class definition. These behaviors are the result of adapting integrity constraints of RDBMS to OODBMS—that is, the referential integrity in RDBMS can be mapped to the relational integrity and the referential integrity. However, the limitation of Jagadish's work is only to suggest the maintenance of integrity, not to provide the behavioral semantics of objects in maintaining the relationship integrity.

Complex objects in ORION OODBMS (Kim, 1987) are the well-known concept in the semantics of static and behavioral relationship of objects. In that work, a complex object represents a 'part–whole' relationship, which uses two kinds of semantics: *exclusiveness* and *dependency*. However, these are limited semantics in the 'part–whole' relationship and insufficient to represent all relationship requirements of various real world applications.

A recent work by Bertino (1998) is similar to our work in the sense that it supports composite object in ODMG standard. However, we not only suggest a relationship model but also implement it over OODBMS SOP, discuss several implementation issues, and provide the corresponding solution to these issues.

In compared to other works, our contributions can be summarized as follows. First, our work deals with a comprehensive part–whole relationship semantics, which includes the various modeling concepts and integrity constraints ever developed in UML, RDBMS and OODBMS areas. Second, in addition to the static part–whole relationship semantics, we provide the complete behavior semantics which are used to maintain the static relationship semantics. Finally, we show that the relationship semantics can be naturally incorporated into the ODMG C++ binding model, and can be implemented.

3. Our relationship model: conceptual design

In this section, we suggest a 'part–whole' relationship model. Please note that we describe its conceptual design. For its ODMG specific binding and examples, see the following section. In the part–whole relationship, an object corresponding to part is called a part-object and an object corresponding to whole is called a whole-object. Traditional approaches to the 'part–whole' relationship provide limited semantics or a system-specific semantic. In contrast, we suggest several kids of relationship so that an application can choose the appropriate relationship types. In addition, this model is applicable on top of ODMG-3.0 compliant OODMBS.

There is a trade-off between the complexity of a model and the efficiency and simplicity of its usage. Because of the differences in the meaning or behavior of relationships required in various application fields, it is difficult to define a model that can support all kinds of semantics. From reviewing of previous papers (Jagadish, 1992; Kim, 1987; Peckham et al., 1995), we decided that the part–whole relationship could be systematically defined using following three dimensions: *exclusiveness*, *multiplicity*, and *dependency*.

The *exclusiveness* means whether a part-object can simultaneously be the part object of several different whole-objects. The *multiplicity* indicates how many objects can participate in the relationship. There are two types of multiplicity in the 'part–whole' relationship: the number of part objects that a whole object can have and the number of whole objects that a part object can have. The *dependency* represents whether the existence of an object is dependent on another object. The dependency also can be divided into two types: whether the existence of a part-object depends on the existence of a whole-object and whether the existence of a whole-object depends on the existence of a part-object. In the following, we give the detailed explanation on each dimension.

The *exclusiveness* has the following three options—Global-Exclusive, Local-Exclusive, and Fully-Shared. Global-Exclusive (GE) is exclusive among different relationships. A part-object with a GE relationship can belong to a whole-object using only that relationship. That is, an object cannot be a part-object through different relationships at the same time. Like (a) in Fig. 1, a student cannot be both the Master Degree student and the PhD Degree student. Local-Exclusive (LE) is exclusive among same relationships. A part-object with LE relationship can belong to only a single whole-object through that relationship. Like (b) in Fig. 1, a monitor cannot belong to two computers through the relationship r5 with LE semantics. However, a printer can simultaneously belong to two computers through r6 because it does not have LE semantics. Fully-Shared (FS) represents a relationship that has neither GE nor LE semantics and means that a part-object with FS relationship

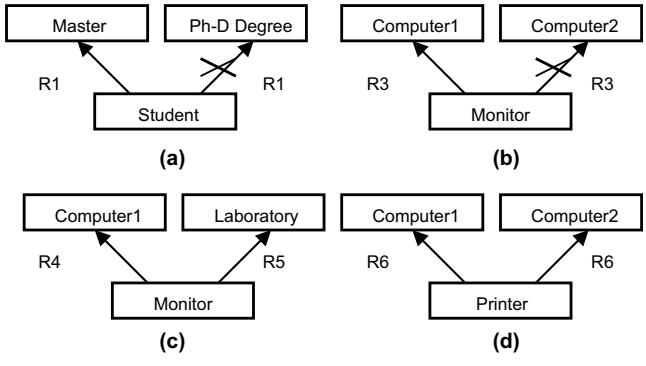

Fig. 1. Example of exclusiveness.

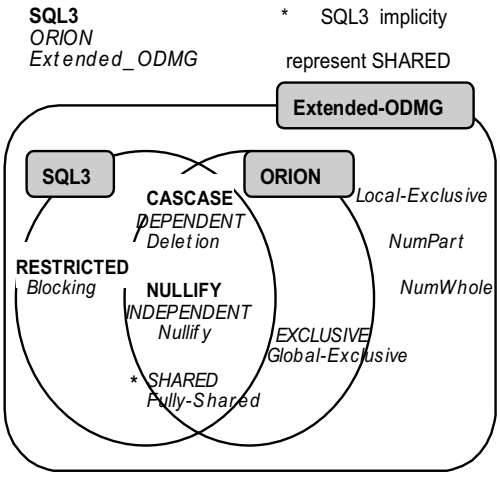

Fig. 2. Expressive power of the relationship of SQL3, ORION, and extended ODMG.

can be belong to several whole-objects through other relationships. GE and LE are orthogonal semantics. In Fig. 1, a monitor has a non-GE relationship with a laboratory and also a LE relationship with a computer.

The *multiplicity* represents how many whole-objects and part-objects can have a relationship and there are two elements—NumPart and NumWhole. NumPart represents the maximum number of part objects a whole objects can have and NumWhole represents the maximum number of whole objects a part objects can have. NumWhole is related with the LE of exclusiveness. When NumWhole is 1, it also means the LE in exclusiveness.

The *dependency* represents whether the existence of a whole-object or a part-object depends on the corresponding part-object or whole-object. The dependency has three elements—Deletion, Nullify, and Blocking—and each element is applicable to both a whole-object and a part-object. The semantics of each dependency element is as follows. With Deletion, when a whole-object is deleted or the relationship itself is deleted, its corresponding part-object is also deleted. With Nullify, when a whole-object is deleted or the relationship is deleted, its corresponding part-object survives. With Blocking, when a user tries to delete a whole-object, if its corresponding part-object exists, the deletion is not allowed. For each element, when a part-object is deleted, the same semantic is applied. Fig. 2 compares the expressive powers among the relationship semantics of SQL3 (Horowitz, 1992; Markowitz, 1991; Turker and Gertz, 2001), ORION (Kim, 1987) and our extended model.

4. Our relationship model: its application to ODMG C++ binding and usage examples

In the previous section, we conceptually described our extended model for 'part–whole' relationship. In this section, we will explain how to represent the extended relationship model in the ODMG-3.0 C++ binding. For this, we propose a new relationship class, which inherits the template class d_Rel_Ref from ODMG-3.0 C++ binding. After describing this class, we explain the semantics of our relationship model in detail, from the perspective of a 'part-object' and a 'whole-object'. And we provide a usage example of our new relationship model.

4.1. Extended relationship classes

In C++ binding of ODMG-3.0, a relationship specification names and defines a traversal path for the relationship. This traversal path is represented by template class d_Rel_Ref <class T, const char* M>. A class, which has this template class as a variable, refers to the template class T and the template class T also refers to the class. The variable name in the template class T, which refers to the class, is M.

To effectively represent our extended relationship model in the ODMG-3.0 C++ binding, we propose a new relationship class, which inherits the template class d_Rel_Ref of ODMG C++ binding. Users can use this relationship class as a primitive type in the user defined classes. In addition, we also define other classes, inheriting the template class d_Rel_Set, d_Rel_List, and d_Rel_Bag so as to represent the relationship about collection. The advantages of this implementation technique are that (1) referential integrity is automatically supported and (2) advantages of d_Rel_Ref, which inherits object handler d_Ref, are also preserved. Because a relationship is implemented as a class, a relationship can represent behavior semantics, which can control the behavior of other objects involved in the relationship, in addition to the effective representation of structural relationship among objects.

In this paper, we propose two kinds of extended relationship type: a relationship type to indicate a part-object from a whole-object side and a relationship type to indicate a whole-object from the part-object sides. For these types, we define two new classes inheriting

```
template<class T, const char* Member, const char* Option>
class d_ Part_Ref: public d_Rel_Ref<T, Member> {
public :
   d_Part_Ref();          // constructor
  ~d_Part_Ref();          // destructor

   d_Part_Ref<T, Member, Option>& operator =(Ref<T>& from);
   d_Part_Ref<T, Member, Option>& operator =(void* from);
   void clear();
   void destroyobj();
};
```

Fig. 3. Class d_Part_Ref.

from the class d_Rel_Ref in ODMG-3.0 C++ binding, that is, class d_Part_Ref and d_Whole_Ref. Fig. 3 shows the declaration of class d_Part_Ref. Class d_Whole_Ref also has the similar declaration.

Similarly, as shown below, class d_Part_Set and class d_Part_List are inherited respectively from d_Rel_Set (a set type collection class in ODMG) and d_Rel_List (a list type collection class in ODMG). They are used to represent several part-objects in a whole-object through a relationship.

```
template <class T, const char* Member, const char*
Option, const char* Max> class d_Part_Set : public
d_Rel_Set <T, Member> {...}
template <class T, const char Member, const char*
Option, const char* Max> class d_Rel_List : public
d_Rel_List <T, Member> {...}
```

Class d_Whole_Ref is also inherited from class d_Rel_Ref. This class is used to represent a whole-object related with a part-object. Both class d_Whole_Set and class a_Whole_List are used to represent several whole-objects related with a part-object.

```
template <class T, const char* Member, const char*
Option> class d_Whole_Ref : public d_Rel_Ref <T,
Member> {...}
template <class T, const char Member, const char*
Option, const char* Max> class d_Whole_Set : pub-
lic d_Rel_Set <T, Member> {...}
template <class T, const char Member, const char*
Option, const char* Max> class d_Whole_List : pub-
lic d_Rel_List <T, Member> {...}
```

In the above, the argument const char* Member, as in ODMG-3.0 d_Rel_Ref, is used to indicate the name of a relationship variable in a corresponding object. The argument const char* Option is used to specify the semantic of the relationship. Finally, the argument const char* Max is optional only for collection classes and is used to represent the multiplicity.

Fig. 4 shows the inheritance hierarchy between the relationship classes in ODMG-3.0 and our new relationship classes.

4.2. The semantic of the relationship type in whole-objects

In this section, we explain how three elements of our relationship model—*exclusiveness*, *multiplicity*, and *dependency*—are represented in the relationship type in a whole-object.

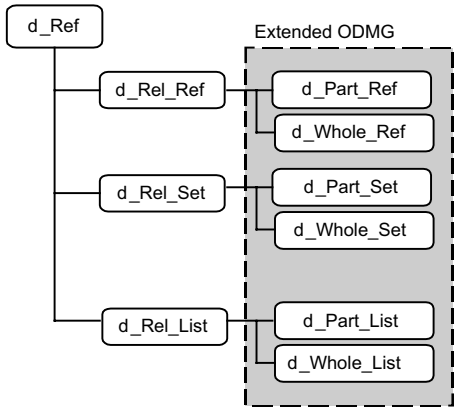

Fig. 4. Class hierarchy of relationship classes.

Because *exclusiveness* is applied to both a whole-object and a part-object, GE is represented in the relationship of whole-objects and LE is represented in the relationship of part-objects. The reason that these two semantics are represented in other objects is that the semantics of GE and LE are orthogonal to each other and a part-object is more suitable to represent LE, which is related with *multiplicity*. LE is represented either in a class d_Whole_Ref that has a single a whole-object or in a collection class that has the value one as maximum number of whole objects. *Multiplicity* and *dependency* must be represented both in a whole-object and in a single object. In a relationship type of a whole-object, multiplicity means the number of part-objects a whole object can have, and *dependency* means the effect of existence of a whole-object to a part-object. *Multiplicity* is represented with the template argument MAX of d_Part_Set and d_Part_List. *Exclusiveness* and *dependency* are represented with the template argument Option of d_Part_Ref, d_Part_Set, and d_Part_List. Option can have the six values, as shown in Table 1.

Table 1 is the combination of dependency and exclusiveness, where exclusiveness can have one of two meanings, GE and Non-GE. This is because the relationship from a whole-object has only semantic GE among three semantics of exclusiveness. The semantic Non-GE is symbolized as Shared. Each Option representing the semantic of a relationship type is as follows. (Because this semantic is commonly applied to d_Part_Ref, d_Part_Set, and d_Part_List, we will explain only the d_Part_Ref case.)

Table 1
The semantic of the relationship type in whole-objects

	Global-Exclusive	Non-Global-Exclusive
Deletion	ED (exclusive deletion)	SD (shared deletion)
Nullify	EN (exclusive nullify)	SN (shared nullify)
Blocking	EB (exclusive blocking)	SB (shared blocking)

- ED (exclusive deletion) has two meanings. First, a part-object belonging to a whole-object through ED semantic can participate in only one part–whole relationship. This is related to the case when a whole-object tries to refer a part-object. When a whole-object tries to refer a part-object through a relationship R1, which already belongs to another whole-object through other relationship R2, it cannot refer that part-object. Second, the existence of a part-object with ED semantic is dependent upon the existence of a whole-object. If a whole-object or the whole-object's d_Part_Ref relationship is deleted, part-objects related with the whole-object or the relationship is also deleted.
- SD (shared deletion): When a whole-object tries to have a part-object through SD semantic, which already belongs to another whole-object through other relationship, the whole-object can take the part-object as its part. If a part-object already belongs to other whole-objects through d_Part_Ref either with ED, EN, or EB semantics, it cannot be referenced. In the case when the part-object already belongs to other whole-object through a d_Part_Ref either with SD, SN, or SB relationship semantics, it can be referenced. If a whole-object or the whole-object's d_Part_Ref relationship is deleted, it needs to be checked whether each of its part-objects also belongs to other whole-objects. If each part-object does not belong to any other whole-objects, it is also deleted.
- EN (exclusive nullify): First, a part-object belonging to a whole-object with EN semantic can participate in only one part–whole relationship. When a whole-object tries to have a part-object through relationship R1 with EN semantic and the part-object already belongs to another whole-object through other relationship R2, it cannot be referenced. Second, when a whole-object or a d_Part_Ref relationship with EN semantic is deleted, its part-objects are not deleted.
- SN (shared nullify): First, a part object which already belongs to a part–whole relationship can belong to a whole-object through another relationship with SN semantic. However, it is limited to the case when a part object has belonged to the other d_Part_Ref with either SD, SN, or SB semantic. Second, when a whole-object or a d_Part_Ref relationship with SN semantic is deleted, its part-objects are not deleted.
- EB (exclusive blocking): If a whole-object has a relationship with EB semantic, its part-objects can have only one part–whole relationship at a run-time and it cannot be deleted when it has a part-object with EB semantic. Therefore, when a whole-object tries to have a part-object through a relationship R1 with EB semantic and the part-object has already belonged to (another) whole-object through other relationship R2, it cannot be referenced. When a whole-object is deleted, it can be deleted only after all its part-objects with EB semantic are deleted.
- SB (shared blocking): If a whole-object has a relationship with SB semantic, it can take, as its part, an object already belonging to other part–whole relationships and the whole-object cannot be deleted if it has any part-object with SB semantic. Therefore, a part object that belongs to a d_Part_Ref relationship with either SD, SN, or SB semantic, can also belong to another whole-object through a relationship with SB semantic. When a whole-object is deleted, it can be deleted only after all its part-objects with SB semantic are deleted.

4.3. The semantic of the relationship type in part-object

In a part-object, all semantics of *multiplicity* and *dependency* to a whole-object can be used. However, for *exclusiveness*, only LE semantic can be used. *Multiplicity* represents the number of whole-objects to which a part-object can belong. *Dependency* represents the effect of a part-object's existence to a whole-object. We explained in Section 4.2 how the semantic LE of *exclusiveness* is represented in a part-object. *Multiplicity* is represented as template argument MAX of collection type d_Whole_Set and d_Whole_List. Similarly, *dependency* is represented as template argument Option of d_Whole_Ref, d_Whole_Set and d_Whole_List. The Option of *dependency* can be one of three elements in Table 2.

The semantics of each Option element is as follows. Again, we will explain the semantics in the case of d_Whole_Ref. In the case of d_Whole_Set and d_Whole_List, the semantics are same to d_Whole_Ref case.

- DT (DeleTion): When a part-object is deleted, its whole-object(s) through the DT semantic must be also deleted. That is, a whole-object cannot exist alone without the part-object.
- BK (Blocking): When a part-object is deleted, it cannot be deleted if it has any BK relationships to whole-objects. That is, the part-object can be deleted only when the BK semantic relationship or a whole object related with this relationship is deleted.
- NK (Nullify): A part-object can be deleted regardless of its relationships with whole-objects. That is, when a part-object is deleted, the whole object is not deleted.

Table 2
The semantics of the relationship type in part-objects

Deletion	Nullify	Blocking
DT	NF	BK

4.4. An example using the extended relationships

Fig. 5 shows a schema declaration using the extended relationship type proposed in this paper. This schema is for a Computer with a part-object Monitor. A whole-object, Computer, has a part-object, Monitor, through relationship type d_Part_Ref with semantic ED. The relationship type d_Part_Ref means that a Computer can have only one part-object through this relationship. Semantic ED means that the Monitor cannot be the part-object of other whole-objects through other relationship and when the Computer is deleted, the Monitor is also deleted. The part-object, Monitor, belongs to a whole-object, Computer, through relationship type d_Whole_Ref with semantic NF. The relationship type d_Whole_Ref means that the Monitor cannot belong to more than one whole-object. Semantic NF means that deletion of the Monitor does not affect the existence of the Computer.

Fig. 6 shows the behavior of objects with part–whole relationships of the example schema in Fig. 5. An object myPC and an object yourPC have the type of class Computer. An object monitorObj has the type of class Monitor. Fig. 6(a) has no part–whole relationship between the object myPC and the object monitorObj. Figure (b) shows a state that the object monitorObj becomes a part object of object myPC using the assignment operator = as follows:

myPC.monitor = monitorObj; (code 1)

When the code 1 runs in application program, the database automatically assigns myPC into the variable computer of the object monitorObj.

Fig. 6(c) shows a state when an object yourPC tries to make the object monitorObj its part-object using the following code:

```
extern const char _ED[], _NF[];
extern const char _monitor[], _[]mputer[];
class Computer : public PObject   {
public:
d_Part_Ref<Monitor, _computer, ED> monitor;
};
class Monitor : public PObject {
public :
d_Whole_Ref<Computer, _monitor, NF> computer;
};
const char _ED[] = "ED";
const char _NF[] = "NF";
const char _monitor[] = "Monitor";
const char _computer[] = "computer";
```

Fig. 5. A schema using the extended relationship.

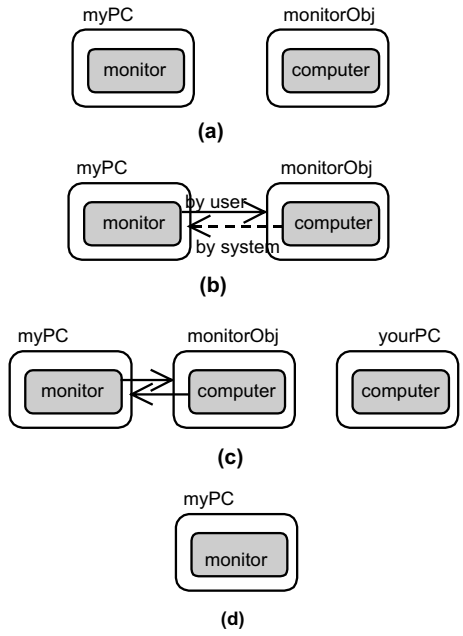

Fig. 6. An example using our extended relationship.

yourPC.monitor = monitorObj; (code 2)

When the code 2 runs, it does not make any change. The object yourPC cannot have the object monitorObj as its part-object because the relationship semantic between two classes is ED and the object monitorObj is already part-object of other object.

If the object myPC deletes its part–whole relationship with the object monitorObj using function clear() as shown in the following code 3, the part-object monitorObj is also deleted. The result is shown in Fig. 6(d):

MyPC.monitor.clear(); (code 3)

When the code 4 runs, the database automatically runs the code 5 without user intervention. Therefore, the object monitorObj is automatically deleted.

myPC.destroyObj(); (code 4)

monitorObj.destroyObj(); (code 5)

If a user want to represent, without using our extended relationship type, the semantic such as ED, (s)he should hard-code it within application programs. The disadvantages of hard coding are that it cannot explicitly represent the relationship within objects and that it requires the modification of all the related codes when the semantic need to be changed. On the other hand, using our extended relationship type, users need to just change the Option field of the extended relationship declaration.

5. Implementation of extended relationship types

In this section, we explain two technical issues encountered when implementing our extended relationship

types in ODMG-3.0 C++ binding, and propose our solutions. The first one is how to exploit the destructor and assignment operator to implement the relationship model in ODMG standard. The second one is about the anomalies which may arise due to the propagation of relationship operations.

5.1. Destructor and assignment operator

Object deletion and assignment operators are very important interfaces in the extended relationship type. In ODMG C++ binding, each operator is implemented as a destructor and an assign operator of a relationship class.

Before we explain the implementation of the destructor and the assign operator, we explain how the information about the relationship type is registered into the schema manager of Soprano repository system. To support the extended relationship type in the database, the following information about a whole-object and a part-object must be accessed at runtime: Has an object O any part-object? If the object O has a part-object, what is the semantic of the relationship? And does the object O belongs to other whole-objects? Such information is registered in the schema manager during schema import. The schema manager manages all the information about the registered classes and their members/member functions. For example, if a schema in Fig. 5 is registered, the schema manager has, for class Computer, the following information about its member d_Part_Ref <Monitor, _computer, ED> monitor.

- Member name: monitor
- Pointer to the data structure of member's domain class: Monitor
- Relationship type: d_Part_Ref
- Name of a relationship variable of corresponding part object: computer
- Semantic of relationship: ED

This information in the schema manager is exploited by the destructor and the assign operator. Algorithms 1 and 2 show the destructor and the assignment operator of the class d_Part_Ref, respectively.

- Destructor: When an object myPC of the class Computer is deleted, its part-object Monitor also must be deleted. Deletion of its part-object is implemented in the destructor of the class d_Part_Ref, because, when an object myPC is deleted, the destructor of the class Computer is called and the destructor also calls the destructor of class d_Part_Ref. Algorithm 1 shows the destructor of the class d_Part_Ref. In Algorithm 1, the case of EB and SB in Option is not implemented. The reason will be explained in Section 5.2.

Algorithm 1. The destructor of the class d_Part_Ref

```
template <class T, const char* Member, const char* Option>
d_Part_Ref <T, Member, Option> :: ~d_Part_Ref()
{   :
  IF (Option is 'ED') {
    Delete a part-object
  } ELSE IF (Option is 'SD') {
    IF (a part-object has the relationship d_Whole_Ref and the relationship does not have other objects) {
     Delete a part-object
    }
  }
}
```

- Assignment operator: When an object is assigned to a whole-object as its part-object, it must be made sure that the object can be a part-object of more than one whole-object. Algorithm 2 is the implementation of the operator = in the class d_Part_Ref in Fig. 2.

Algorithm 2. operator = of the class d_Part_Ref

```
template <class T, const char* Member, const char* Option>
d_Part_Ref <T, Member, Option> & d_Part_Ref <T, Member, Option>
:: operator = (Ref <T> & from)
{   :
  IF (Option is not 'ED', 'EN' and 'EB') {
    IF (an object 'from' is not a part-object of other whole-objects) {
     take an object 'from' as a part-object
    }
  } ELSE {
    IF (an object 'from' does not belong to a whole-object through the semantic 'ED', 'EN', or 'EB' relationship) {
     take an object 'from' as a part-object
    }
  }
}
```

5.2. The propagations of relationship operations, anomalies, and solutions

Because the operation of a part–whole relationship is propagated to other object, a ill-designed schema may bring some unexpected results, that is, anomalies, at runtime. The similar situation may arise also in primary and foreign key relationship in RDBMS, that is, the cascaded propagations of the foreign keys' behavioral actions (Markowitz, 1991). To attack this problem, SQL3 (Markowitz, 1991) restricts the order of operations. For the similar anomalies in OODBMS environ-

ment, Peckham et al. (1996) suggested a solution which notifies the possible anomalies to users when a schema may cause problems.

Among the relationship's semantics defined in this paper, dependency has the possibility to bring anomalies. When Deletion and Blocking of Dependency and Shared of Exclusiveness is used together, the following problems arise.

- Anomaly 1: in case of deletion of part-objects

  Let us consider Fig. 7(1). When a user tries to delete an object W1, the whole-object W1 and its part-objects should not be deleted because the object W1 has the relationship of the semantic SB. However, when the object W1's destructor is called, the relationship's destructor with the semantic SD is first called and next the relationship's destructor with the semantic SB is called. Therefore, even though the whole object W1 and a part-object P2 is not deleted, a part-object P1 is deleted.
- Anomaly 2: in case of different orders of operation executions

  Let us consider Fig. 7(2). When a user deletes an object A, the result of this operation may differ according to the calling order of part-object's destructor. If the destructor of the SD relationship is called first, the calling sequence of cascaded destructors is following: A's destructor, B's destructor, and C's destructor. The destructor of the relationship with the SB semantic of the object A is called after the object C is deleted. Therefore, all three objects are deleted. However, if the relationship's destructor with the SB semantic is called first, the object A cannot be deleted because of the object C. Therefore, the destructor of the A's SD relationship is not called and all three objects are not deleted. This happens because, in C++ language environment, the calling sequence of part-objects' destructors becomes different according to the order of the relationship variable declaration in a class.

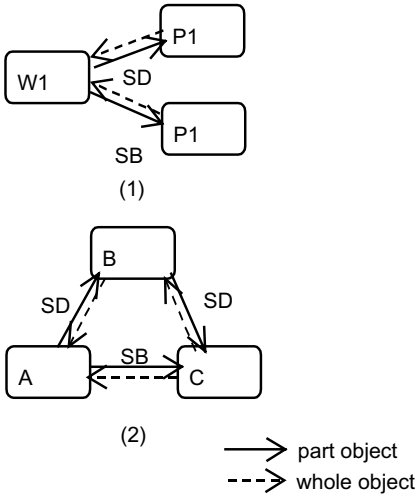

Fig. 7. Anomalies in the propagation of relationship operations.

To solve these problems, we decide to call the destructor of relationship with the semantic SB before other destructors. In Fig. 7(1), when an object W1 is deleted, anomaly 1 will not happen if the destructor of relationship with semantic SB is called first. In Fig. 7(2), when an object A is deleted, all three objects will not be deleted if the destructor of the SB is called first.

To implement this, the calling order of the destructors of an object's members should be changed appropriately. In C++, the calling order of the destructor of an object's member is reverse to the declaration order of an object's members (Stroustrup, 1997). However, we cannot control the calling order of the destructor in C++. So, we implemented in the way that the destructor of the SB relationship is called before the destructor of the object itself. Therefore, the destructor explained in Section 5.1 do nothing for the relationship with the semantic SB. Instead, a function destroyobj(), in which an object is deleted, finds the relationship with the semantic SB before it calls delete, which actually deletes the object. If the relationship with the semantic SB exists and it points to a part-object, the function destroyobj() does not delete the object by returning without calling the delete function. Algorithm 3 shows the function destroyobj().

Algorithm 3. Function destroy()

```
int RefAny :: destroyobj(void)
{   :
   FOR (the relationship variable) {
      IF (the semantic of the relationship is 'SB' or
      'EB') {
       IF (the relationship point to the object itself) {
         Return ErrorMsg;
       }
      }
   delete the object;
}
```

For the relationship with the semantic EB, its deletion is implemented in the function destroyobj() instead of in the its destructor because of convenience of implementation. For the d_Whole_Ref relationship with the semantic BK, its deletion is implemented in the function destroyobj() similar with the d_Part_Ref relationship with the semantic SB.

The schema manager has the information of the inherited members as well as its local members. When a parent class has the relationship with the semantic SB and its child class has the relationship with the semantic SD, the calling of the child class's function destroyobj() returns without calling the delete because it finds the relationship with the semantic SB.

6. Conclusion

In this paper, we explained the design and implementation of a part–whole relationship, which extends the relationship types in ODMG-3.0 C++ binding. Our part–whole relationship model is implemented on top of SOP OODBMS (Ahn et al., 1996). The fundamental rule in designing our relationship model is to provide a set of relationship types which is compliant to ODMG-3.0 standard and which is simple and flexible for users to choose the appropriate relationship features for their applications. The part–whole relationship model is represented with exclusiveness, deletion, and multiplicity for both a whole-object and a part-object. Another contribution of this paper is that we identified some anomalous effects of uncontrolled sequence of relationship operations, and provided solutions for them.

References

Ahn, J.H., Lee, K.W., Song, H.J., Kim, H.J., 1996. Soprano: design and implementation of an object storage system. Journal of Korean Information Society (C) 2 (3), 243–255.

Albano, A., Ghelli, G., Orsini, R., 1991. A relationship mechanism for a strongly typed object-oriented database programming language. In: Proceedings of the 17th International Conference on Very Large Data Bases. pp. 565–575.

Bernstein, P.A., 1998. Repositories and object oriented databases. SIGMOD Record 27 (1), 88–96.

Bertino, E., 1998. Extending ODMG object model with composite objects. In: Proceedings of 1998 ACM International Conference in Object-Oriented Programming: Systems, Languages, and Applications. pp. 259–270.

Cattell, R.G.G., Barry, D.K., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., Velez, F., 2000. The Object Data Standard: ODMG 3.0. Morgan Kaufmann Publishers, San Francisco, CA.

Codd, E.F., 1970. A relational model for large shared databases. Communications of the ACM 13 (6), 377–387.

Davis, H.C., 1998. Referential integrity of links in open hypermedia systems. HyperText 98, 207–216.

Halper, M., Celler, J., Perl, Y., 1994. Integrating a part relationship into an open OODB system using metaclasses. In: Proceedings of the 3rd International Conference on Information and Knowledge Management. pp. 10–17.

Halper, M., Celler, J., Perl, Y., 1998. An OODB part–whole model: semantics, notation and implementation. IEEE Transaction on Data and Knowledge Engineering 27 (1), 59–95.

Horowitz, B.M., 1992. A run-time execution model for referential integrity maintenance. In: Proceedings of the 18th International Conference on Very Large Data Bases. pp. 548–556.

Jagadish, H.V., 1992. Integrity maintenance in an object-oriented database. In: Proceedings of the 18th International Conference on Very Large Data Bases. pp. 469–480.

Kim, W., 1987. Composite object support in an object-oriented database system. In: Proceedings of the 1987 ACM International Conference in Object-Oriented Programming: Systems, Languages, and Applications. pp. 118–125.

Lochovsky, F., 1985. Special issue on object-oriented systems. Database Engineering Bulletin 8 (4).

Markowitz, V.M., 1991. Safe referential integrity structures in relational databases. In: Proceedings of the 17th International Conference on Very Large Data Bases. pp. 123–132.

Peckham, J., MacKellar, B., Doherty, M., 1995. Data model for extensible support of explicit relationships in design databases. The VLDB Journal 4 (2), 157–191.

Peckham, J., Maryanski, F., Demurjian, S.A., 1996. Toward the correctness and consistency of update semantics in semantic database schema. IEEE Transaction on Knowledge and Data Engineering 8 (3), 503–507.

Rumbaugh, J.E., 1987. Relations as semantic constructs in an object-oriented language. In: Proceedings of the 1987 ACM International Conference in Object-Oriented Programming: Systems, Languages, and Applications. pp. 466–481.

Rumbaugh, J.E., Jacobson, I., Booch, G., 1999. The Unified Modeling Language Reference Manual. Addison-Wesley Publisher, Co.

Stroustrup, B., 1997. The C++ Programming Language. Addison-Wesley Publisher, Co.

Turker, C., Gertz, M., 2001. Semantic integrity support in SQL:1999 and commercial (object-)relational database management systems. The VLDB Journal 10 (3), 241–269.

W3C. World Wide Web Consortium, 2001. XML Linking Language (XLink). Available from <http://www.w3.org/TR/xlink>.

W3C. World Wide Web Consortium, 2001. XML Base. Available from <http://www.w3.org/TR/xmlbase>.