O2-SE
f-papers

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING

A publication of the IEEE Computer Society

NOVEMBER/DECEMBER 2002 VOLUME 14 NUMBER 6 ITKEEH (ISSN 1041-4347)

REGULAR PAPERS

Agent-Based Systems

Artificial Intelligence

Databases

Data Mining and Discovery

Data Mining and Knowledge Discovery

Data Structures and Algorithms

Distributed Databases

Information Retrieval

Knowledge Base

Knowledge-Based Systems

(Contents continued on back cover)

AUTO**3-DIGIT 331
Naphtali D. Rishe
FLORIDA INTERNATIONAL UNIVERSITY 358 43 31
HIGH-PERFORMANCE DATABASE RESEARCH CENTER
11200 S.W. 8TH STREET
MIAMI FL 33199-0001

IEEE COMPUTER SOCIETY
IEEE
http://computer.org
tkde@computer.org

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 14, NO. 6, NOVEMBER/DECEMBER 2002

Specifying and Enforcing Association Semantics via ORN in the Presence of Association Cycles

Bryon K. Ehlmann, *Member, IEEE Computer Society*, Gregory A. Riccardi, *Member, IEEE*,
Naphtali D. Rishe, *Member, IEEE Computer Society*, and Jinyu Shi

Abstract—Object Relationship Notation (ORN) is a declarative scheme that allows a variety of common relationship types to be conveniently specified to a Database Management System (DBMS), thereby allowing their semantics to be automatically enforced by the DBMS. ORN can be integrated into any data model that represents binary associations or DBMS that implements them. In this paper, we give a brief description of ORN syntax and semantics and provide algorithms that can be used to implement ORN. These algorithms must deal with the presence of association cycles in the database. We explore in detail the problems caused by such cycles and how ORN and its implementation deal with them, and we show that ORN semantics are noncircular and unambiguous.

Index Terms—ORN, relationship semantics, association cycle, data modeling, object databases, complex objects.

◆

1 INTRODUCTION

The Object Relationship Notation (ORN) is a declarative scheme for defining a variety of common relationship types, i.e., the "is part of," "is defined by," "is owned by," and "is associated with" types of relationships and their many variations. These relationships are termed *associations* in the Unified Modeling Language (UML) [1], define the *class-composition hierarchy* in an object database [2], and are the glue that binds together a *complex object*.

A complex object is a collection of closely interrelated objects whose associations are often constrained. It is typical of such objects that the lack of or removal of related objects or association instances, i.e., *links*, may violate the object's integrity. The benefit of ORN is that it allows database designers to define the proper bindings between the components of complex objects, and allows the Database Management System (DBMS) to enforce these bindings.

ORN can be used during system analysis and design to capture and document in a data model the semantics of complex object associations. The same notation can then be used during implementation to define these semantics to the DBMS. This allows the early detection of association subtleties and inconsistencies and the automatic maintenance of consistent association semantics by the DBMS, thereby improving database integrity. Significantly, this is achieved without programming or without the specification of complex SQL constraints and triggers [3], [4].

In a previous paper [5], ORN was compared to other declarative schemes for specifying association semantics—those proposed for various object models [6], [7], [8] as well as the REFERENCES clause of SQL [3], [9]. The comparison revealed that the most unique aspect of ORN and what accounts for its ability to specify a larger variety of association types is that it provides for the enforcement of upper and lower bound cardinality constraints and allows delete propagation based on these constraints. It is noteworthy that to our knowledge none of the declarative schemes for object models proposed in the late 1980's and early 1990's have been adopted in commercial DBMSs, and little work in this area has occurred since then. This is regrettable since a significant increase in productivity can result from having a powerful declarative capability, like ORN, for specifying association semantics.

Other papers have explored various aspects of ORN. In [10], an integrated methodology based on ORN is presented for developing associations in a database. The paper shows how ORN, unlike the declarative scheme of SQL, can be incorporated into ER-like Diagrams [11]. Hardeman [12] shows how, with ORN, subtleties and inconsistencies in association behavior can be identified and automatically detected during analysis and design. Ehlmann and Riccardi [13] discuss an extensible, ODMG-93 compatible [14] Object DBMS prototype, called Object Relater *Plus* (OR+), which implements ORN as an extension to Object Store [15]. Ehlmann [16] presents the features and benefits of the ORN Simulator, a prototype database modeling tool, which is supported by OR+ and available on the Web [17]. A formal specification of ORN semantics is given in [18]. Ehlmann and Yu [19] discuss the integration of ORN into UML class diagrams and, finally, Ehlmann and Stewart [4] describe the

---

- B.K. Ehlmann is with the Department of Computer Science, Southern Illinois University at Edwardsville, Edwardsville, IL 62026. E-mail: behlman@siue.edu.
- G.A. Riccardi is with the Department of Computer Science, Florida State University, Tallahassee, FL 32306. E-mail: riccardi@cs.fsu.edu.
- N.D. Rishe is with the High-Performance Database Research Center, Florida International University, 11200 S.W. 8th Street, Miami, FL 33199. E-mail: rishen@cs.fiu.edu.
- J. Shi is with Bank of America, 9000 Southside Blvd., Jacksonville, FL 32256. E-mail: jinyu.shi@bankofamerica.com.

Manuscript received 10 Aug. 1999; revised 21 Dec. 2001; accepted 12 July 2002.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 110404.

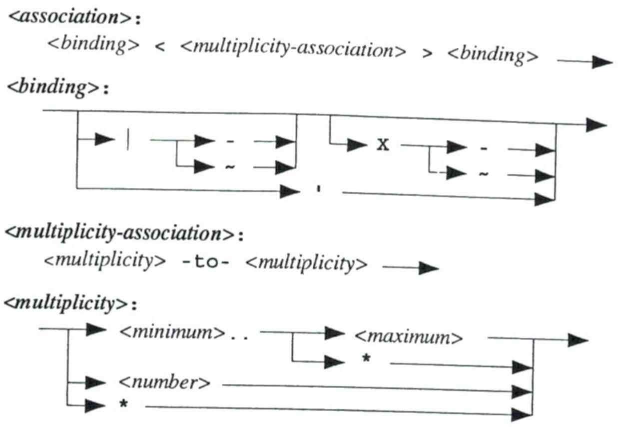

Fig. 1. ORN syntax.

syntax, semantics, and pragmatics for incorporating ORN into SQL as well as the benefits.

The primary contribution of this paper is to present the algorithms used to implement ORN semantics. They are given at an object-association level of abstraction and outline the code that translation tools must generate to implement ORN. In OR+, this code is implemented as methods on abstract, persistent classes. In a relational database system, the code would be implemented, at least partially, as constraints and triggers on related tables.

A secondary contribution is to show that ORN semantics as implemented by these algorithms are noncircular and unambiguous, in spite of association cycles. An *association cycle* occurs in a database when a object is related to itself, directly or indirectly. The problems posed by such cycles in specifying ORN semantics—infinite and alternative processing paths, which can result in circularity and ambiguity—are inherent in any scheme that defines association semantics recursively, as does ORN.

The remainder of this paper is organized as follows: We first briefly describe ORN syntax and semantics in Section 2. (A more detailed description can be found in [18].) In Section 3, we present and explain the algorithms used in OR+ to implement ORN semantics, and in Section 4, we explore their operation in the context of association cycles. We conclude the paper in Section 5 with some summary remarks. An appendix provides a proof that ORN semantics are independent of the order in which associations are processed and are therefore unambiguous, provided one specific type of specification is restricted.

2 DESCRIPTION OF SYNTAX AND SEMANTICS

The syntax and semantics of ORN define a taxonomy of binary associations, i.e., association types, that are common to databases. In a nutshell, Fig. 1 gives the syntax of ORN and Table 1 gives its semantics. Fig. 2 shows how ORN is incorporated into a UML class diagram.

Modeled in this diagram is an association between employees and car pools. This association is often used for illustration in the remainder of this paper. An employee may belong to a car pool and a car pool is defined by at least two riders, without which there would be no car pool.

TABLE 1
Meaning of ORN Symbols

< > - Distinguish an *<association>* from a *<multiplicity-association>*

Multiplicity Symbols:

<minimum> - integer ≥ 0 *<maximum>* - integer > 0
`..` - "to" as in `2..6`, meaning two to six
`..*` - "to many" (unbounded) as in `1..*`, meaning 1 to many
<number> - integer > 0, same as *<number>*`..`*<number>*
`*` - "many" (unbounded), same as `0..*`

Binding Symbols:

Binding symbols are described in terms of an object class *C* in an association *A* having the given binding. Deletion of a *C* object succeeds only if all existing association links involving that object are implicitly destructible, i.e., can be *cut*. Also, the deletion of a *C* object or explicit destruction of an *A* link succeeds only if all required implicit deletions succeed. The "none" given below indicates no applicable binding symbol is given.

none - Default implicit destructibility binding. On delete of a *C* object, an existing *A* link is implicitly destructible provided implicit destruction does not violate the multiplicity of *C*.*

|- - Minus implicit destructibility binding. On delete of a *C* object, an existing *A* link is never implicitly destructible. *Implicit* link destruction is denoted by the |, symbolizing a *cut* in the link.

|~ - Propagate implicit destructibility binding. On delete of a *C* object, an existing *A* link is always implicitly destructible. The related object is implicitly deleted when implicit destruction violates the multiplicity of *C*.

none - Default explicit destructibility binding. An *A* link is explicitly destructible provided explicit destruction does not violate the multiplicity of *C*.*

`x-` - Minus explicit destructibility binding. An *A* link is never explicitly destructible. *Explicit* link destruction is denoted by the `x`.

`x~` - Propagate explicit destructibility binding. An *A* link is always explicitly destructible. The object related to the C object is implicitly deleted when explicit destruction violates the multiplicity of *C*.

' - Prime implicit and explicit destructibility binding. On delete of a *C* object, an existing *A* link is implicitly destructible. An implicit delete is done on the related, i.e., subordinate, object. Also, an *A* link is always explicitly destructible. Again, an implicit delete is done on the subordinate object. The implicit deletion of a subordinate object is required, and thus must succeed, if and only if link destruction, implicit or explicit, violates the multiplicity of *C*.*

* The check for a violation caused by the link destruction is deferred until the end of the current complex object operation.

Fig. 3 shows how ORN is incorporated into the Object Database Definition Language (ODDL) of OR+ [13]. This partial specification defines the employee-car pool association to OR+. The Object Database Manipulation Language (ODML) of OR+ provides for database creation, access, and manipulation based on an ODDL specification.

As shown by Fig. 1 and Table 1, associations in ORN are described on two levels. A *<multiplicity-association>* defines a binary association type solely by classes and *multiplicities*, or cardinality constraints. *Bindings* are then added to both ends of an *<association>* to indicate the level of binding between the related objects. The level of binding determines the implicit and explicit destructibility of association links and whether link destruction can result in the implicit deletion of related objects. Implicit destructibility of associations is important since all existing links involving an object must be implicitly destroyed, or cut, before an object can be deleted. Implicit deletions of related objects, which may result from link destructions, enforce multiplicities and define the extent of complex objects and

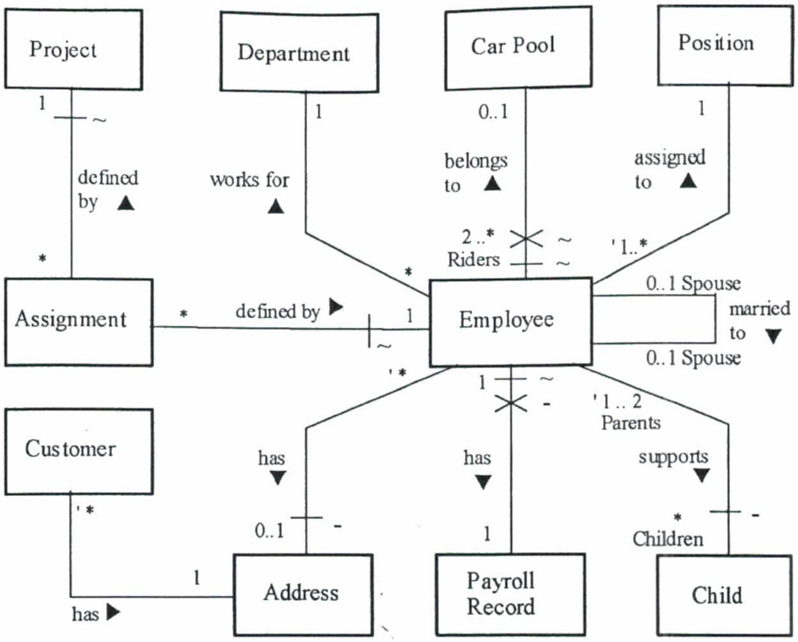

Fig. 2. Class diagram for a company database.

composite objects. Composite objects, or *aggregate objects* in UML terminology, are complex objects whose component objects are more tightly bound by the semantics of an "is a part of" association.

In a <*multiplicity-association*>, the <*multiplicity*> before the **-to-** describes the multiplicity for the *subject* class; the <*multiplicity*> after the **-to-** describes the multiplicity for the *related* class. The subject class multiplicity is the number of objects of the subject class that can relate to a single object of the related class. Likewise, the related class multiplicity is the number of objects of the related class that can relate to a single object of the subject class. For example, the <*multiplicity-association*> for the employee-car pool association is **2..*-to-0..1**. Each object of type employee, the subject class, relates to zero or one car pool. Each object of type car pool, the related class, relates to two to many employees.

In an <*association*>, the <*binding*> before the < indicates the binding for the subject class; the one after the > indicates the binding for the related class. Association semantics are derived from the multiplicity semantics and the semantics of the given bindings. For example, in the <*association*> for employees and car pools, |~X~<2..*-to-0..1>, the |~ symbol of the <*binding*> for the employee subject class means (applying Table 1): On delete of an employee object, an existing employee-car pool link is always implicitly destructible, and the car pool object is implicitly deleted when implicit destruction violates the multiplicity **2..***. The **X~** symbol means: An employee-car pool link is always explicitly destructible, and the related object is implicitly deleted when explicit destruction violates the multiplicity **2..***. The **2..*** multiplicity is violated when the link to the second last employee is destroyed. The default <*binding*> for the related car pool class means (again, applying Table 1): On delete of a car pool object, an existing employee-car pool link is implicitly destructible provided implicit destruction does not violate the multiplicity **0..1**, and an employee-car pool link is explicitly destructible provided explicit destruction does not violate the multiplicity **0..1**. A **0..1** multiplicity is never violated by link destruction.

```
class employee {
    ...
    car_pool        CarPool inverse Riders |~X~<2..*-to-0..1>;
    ...
};
class car_pool {
    ...
    Set<employee> Riders inverse CarPool;
    ...
};
```

Fig. 3. Partial ODDL for the employee-car pool association.

Every association has an inverse where the subject class becomes the related class, and vice versa. The inverse of the employee-car pool association is a car pool-employee association, which can be described as **<0..1-to-2..*>|~X~**.

Below are more of the association semantics involving employees that are defined in Fig. 2 by ORN.

- If an employee is deleted, his link with a department is implicitly destroyed (default binding and * multiplicity), and his assignments are implicitly deleted, as is his payroll record (|~ binding and **1** multiplicity).
- If an employee is deleted, her address is deleted (' binding), unless it is also the address of another employee (|- binding) or customer (default binding and **1** multiplicity). Her position is also deleted (' binding), unless it is also held by another employee (default binding and **1** multiplicity), and all of her children are deleted (' binding), unless a child's other parent also works for the company (|- binding).
- An employee's link to a payroll record can never be explicitly destroyed (X- binding). It can only be

```
Begin nested transaction t on database d
Invoke CreateLink, DeleteObject, DestroyLink, or ChangeLink algorithm
if exception then Abort t
else Commit(t, d)
    if exception then Abort t
```

Fig. 4. Invocation of complex object operation in nested transaction.

destroyed implicitly as a result of the employee's deletion (again, |~ binding and 1 multiplicity).

As can be seen, the association semantics involving an employee object make it a very complex object.

3 IMPLEMENTING ALGORITHMS

Implementation of ORN semantics can be described by algorithms that create objects and association links, delete objects, and destroy and change association links. These operations become complex object operations in the context of ORN. In this section, we describe the impact of ORN semantics on the implementation of object creation and give algorithms for implementing object deletion and link creation, destruction, and change.

When an object of a specific class is created (or instantiated)—e.g., via a primitive object creation operator, like **new** in C++ or Java—the implementation of ORN must ensure that the complex object is properly constructed. In particular, this means that all lower bound multiplicities for any related classes are satisfied before the transaction containing the object creation can commit. In OR+, all classes for which associations are defined are derived from a common base class *d_rObject*, meaning "database relatable object." This class has a constructor that is implicitly called whenever any such object is created. The constructor adds a reference to the object to a set called *LbChecks*, which is associated with the current application-defined transaction. Creation and modification of relatable objects must take place within such a transaction. When it commits, checks are made on each existing object referenced in *LbChecks* to ensure that lower bound multiplicities for related classes are not violated.

Figs. 4, 5, 6, 7, 8, and 9 show algorithms for the other complex object operations. These algorithms provide an abstract view of the actual OR+ implementation of ORN. They do not, for instance, show the details for handling association inheritance. The algorithms are given in a pseudocode where control structure is indicated by indentation. Those given in Figs. 6, 7, 8, and 9 are invoked within a system-supplied nested transaction as shown in Fig. 4. This nested transaction results when a complex object operation is executed within an application-defined transaction. The complex object operations are syntactic variants of *CreateLink*, *DeleteObject*, *DestroyLink*, or *ChangeLink* as defined by their signatures. The nested transaction ensures that these operations are atomic.

```
Algorithm Commit(t: Transaction, d: Database)
/* Commit transaction t on database d.
    for each object x in t.LbChecks – t.Deletes do
        C = type(x);
        for each association A where C is the subject class do
            if lower bound multiplicity for related class of A is violated then
                exit(exception);
    Perform other commit functions;
    if exception then exit(exception);
    if t is a nested transaction then
        Add objects in t.Deletes to Deletes of parent transaction;
    exit(successful);
```

Fig. 5. Algorithm for committing a transaction.

```
Algorithm CreateLink(A: Association, sO: Object, rO: Object,
                     t: Transaction, d: Database)
/* Create a link of type A between subject object sO and related object rO. */
    sC = type(sO);
    rC = type(rO);
    sUb = upper bound multiplicity for sC of A;
    rUb = upper bound multiplicity for rC of A;
    Create link sO ↔ rO of type A;
    if sUb or rUb is violated then exit(exception);
    exit(successful);
```

Fig. 6. Algorithm for creating an association link.

The "Begin" of a transaction initializes two object sets, *Deletes* and *LbChecks*, to empty. These sets are associated with every transaction. *Deletes* is the set of all objects marked for deletion so far by the transaction, including objects marked for deletion by any committed nested transactions. *LbChecks*, discussed previously, is the set of all objects whose association lower bound multiplicities must be checked at the commit, provided that the object is not also in *Deletes*. Fig. 5 describes how these sets are processed when a transaction commits.

The abstract nature of the algorithms given in Figs. 6, 7, 8, and 9 make them independent of a particular implementation, object or relational. A link between objects x and y is represented in the algorithms as an ordered pair $x \leftrightarrow y$,

```
Algorithm DeleteObject(x: Object, t: Transaction, d: Database)
/* Delete complex object x, i.e., x and appropriate related objects as defined
by ORN. Does recursive, depth first traversal of d. t is assumed begun and
initialized on the first, non-recursive call.*/
    if x in t.Deletes or the Deletes of any ancestor transaction then
        exit(successful);
    Insert x into t.Deletes;
    C = type(x);
    for each association A defined where class C is the subject class (in the
            order defined) do
        impB = implicit destructibility binding for class C of A;
        lB = lower bound multiplicity for C of A;
        for each link l = x ↔ rO of type A where x is the subject object
                and rO is a related object do
            Destroy link l;
            case impB
                none: if lB is violated then insert rO into t.LbChecks;
                "|-": exit(exception);
                "|~": if lB is violated then
                          DeleteObject(rO, t, d);
                          if exception then exit(exception);
                ",":  if lB is violated then insert rO into t.LbChecks;
                      Begin nested transaction nT on database d;
                      DeleteObject(rO, nT, d);
                      if exception then Abort nT
                      else Commit(nT, d);
                          if exception then Abort nT;
            end case
        end for
    end for
    Delete primitive object x;
    exit(successful);
```

Fig. 7. Algorithm for deleting an object.

```
Algorithm DestroyLink(l: Link, t: Transaction, d: Database)
/* Explicitly destroy link $l = x \leftrightarrow y$ in d.  t is as defined for DeleteObject. */
    $A$ = type(l);
    Destroy link l;
    for each object o in $x \leftrightarrow y$ do
        $C$ = type(o);  rO = related object of o
        expB = explicit destructibility binding for class C of A;
        lB = lower bound multiplicity for C of A;
        case expB
            none:  if lB is violated then insert rO into t.LbChecks;
            "X-":  exit(exception);
            "X~":  if lB is violated then
                        DeleteObject(rO, t, d);
                        if exception then exit(exception);
            "'":   if lB is violated then insert rO into t.LbChecks;
                   Begin nested transaction nT on database d;
                   DeleteObject(rO, nT, d);
                   if exception then Abort nT
                   else Commit(nT, d);
                        if exception then Abort nT;
        end case
    end for
    exit(successful);
```

Fig. 8. Algorithm for destroying an association link.

where x is the subject object and y is the related object. The type of a link is the association of which it is an instance. The objects of a link, together with its type, make it unique. Every association A has an inverse association, A^{-1}, where the roles of subject and related class are reversed. If $x \leftrightarrow y$ exists as a link of type A, $y \leftrightarrow x$ exists as a link of type A^{-1}. In object database terms, every association is represented by an object-based attribute, e.g., CarPool in class employee, and an inverse object-based attribute, e.g., Riders in class car_pool (see Fig. 3).

In *CreateLink*, Fig. 6, "Create link $sO \leftrightarrow rO$ of type A" means to create the necessary reference(s) between the two objects. In an object database, this involves setting or inserting appropriate references into the object-based attributes of the subject and related objects. For example, to create a link of the employee-car pool association as defined in Fig. 3, the CarPool attribute of an employee must be set to reference a car_pool object and a reference to this employee must be inserted into the Riders attribute of the car_pool object.

In *DeleteObject*, Fig. 7, every association involving the object x is traversed by the outer **for each** loop. In the inner **for each**, a link is implicitly destroyed (by destroying references to and from the related object in an object database) before any implicit delete is attempted on the related object. Thus, the destroyed link is not considered when determining whether or not an implicit delete of the related object is possible. Within the **case** statement, when the lower bound multiplicity is violated on a default, i.e., none, or a ′ binding, an exception does not immediately result; rather, the related object is inserted into the set *t.LbChecks*, deferring any exception until the end of the operation, i.e., the nested transaction commit. This fact will be important to remember in the next section.

The loop in *DestroyLink*, Fig. 8, has two iterations, treating in turn each object in the link as the subject object.

The *ChangeLink* algorithm, Fig. 9, is essentially an explicit destroy of a link for some association A between a subject class object x and related class object y, followed by the creation of a new A link between x and a different object z of the related class. The only difference is that the explicit destructibility binding, *expD*, for the related class is not processed. Lower bound multiplicities for this class will not have been violated since one related class object is simply being replaced by another. Any **X-** binding for the related class will have already been detected since it applies to both ends of an association if given. Any ′ binding for the related class will not result in the implicit deletion of the subject object. In this case, the subject object is simply being made subordinate to a different prime object.

```
Algorithm ChangeLink(l: Link, z: Object, t: Transaction, d: Database)
/* Change link $l = x \leftrightarrow y$ replacing the related object y with z.  t is as defined
for DeleteObject. */
    $A$ = type(l);  $C$ = type(x);

    // Destroy the link between x and y.
    Destroy link l;
    expB = explicit destructibility binding for class C of A;
    lB = lower bound multiplicity for C of A;
    case expB
        none:  if lB is violated then insert y into t.LbChecks;
        "X-":  exit(exception);
        "X~":  if lB is violated then
                    DeleteObject(y, t, d);
                    if exception then exit(exception);
        "'":   if lB is violated then insert y into t.LbChecks;
               Begin nested transaction nT on database d;
               DeleteObject(y, nT, d);
               if exception then Abort nT
               else Commit(nT, d);
                    if exception then Abort nT;
    end case

    // Create a link between x and z.
    uB = upper bound multiplicity for C of A;
    Create link $x \leftrightarrow z$ of type A;
    if uB is violated then exit(exception);
    exit(successful);
```

Fig. 9. Algorithm for changing an association link.

4 ASSOCIATION CYCLES

Others have studied the problems posed by association cycles within relational databases and SQL [20], [21], [22]. In the context of SQL, such cycles are called *referential cycles*, and the concern is not in maintaining cardinality constraints as with ORN, but rather in maintaining referential integrity. Some of this previous work, however, is germane to our exploration of association cycles in this section, where we deal at the entity or object level.

To investigate the problems caused by association cycles, we will study some simple examples of such cycles. Fig. 10 depicts one example. There are just two objects in the database, y1 and z1, and two links, y1 ↔ z1 of association A1 and y1 ↔ z1 of A2. In this and subsequent examples, we assume ORN semantics as implemented by the algorithms given in the previous section and examine what happens when an attempt is made to delete z1.

In Fig. 10, there are two possible scenarios.

- If A1 (or more precisely its inverse $A1^{-1}$) is processed first, trying to delete z1 causes an implicit

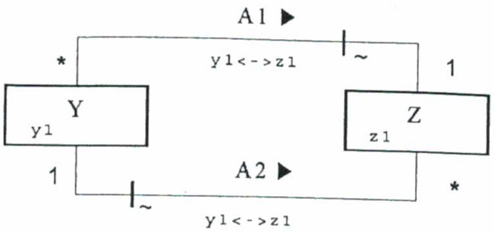

Fig. 10. Association cycle z1 ↔ y1, y1 ↔ z1.

destruction of the y1 ↔ z1 link of A1 (or the z1 ↔ y1 link of $A1^{-1}$) and an implicit delete on y1. This is based on the |~ binding and **1** multiplicity for class Z in the A1 association. The implicit delete of y1 will result in the implicit destruction of the y1 ↔ z1 link of A2 and an implicit delete of z1, which will be successful since z1 has previously been marked for deletion (i.e., the recursive call to *DeleteObject* will exit successful since *x* is already in *t.Deletes*). Thus, the deletion of z1 is successful.

- If A2 is processed first, trying to delete z1 causes an implicit destruction of the y1 ↔ z1 link of A2. Next, A1 is processed, which causes an implicit destruction of the y1 ↔ z1 link of A1 and an implicit delete on y1, which will be successful. Thus, the deletion of z1 is again successful.

One problem with association cycles is that the recursion inherent in the semantics of ORN and often in those of similar declarative schemes is circular unless there is some means to detect an association cycle. As the first scenario above shows, the *DeleteObject* algorithm for ORN detects a cycle and terminates recursion by means of the set *t.Deletes*. Objects are marked for deletion by placing them into this set. Then, recursive propagation of implicit deletes is terminated when an object to be deleted is found in this set, i.e., when an association cycle is detected.

Note that in deleting z1 via the *DeleteObject* algorithm, as described above, the order in which the associations were processed did not matter. Unfortunately, this is not always the case.

Figs. 11 and 12 depict two more association cycles. Fig. 11 is a simplified nonrelational version of an example given in [20]. For both figures, we again examine what happens when an attempt is made to delete z1. In Fig. 11, the "?" indicates a possible implicit destructibility binding. We look at two cases.

For Fig. 11, Case 1, assume the "?" is replaced by a |- binding. Again, there are two scenarios.

- If A1 is processed first, trying to delete z1 causes an implicit destruction of the y2 ↔ z1 link of A1 and an implicit delete on y2. This will be successful and result in the implicit destruction of the y2 ↔ z1 link of A2. Now, when A2 is processed for z1 to see if links exist that require implicit destruction, none is found. Thus, the delete of z1 is successful.
- If A2 is processed first, trying to delete z1 will be unsuccessful because the |- binding prevents the destruction of the y2 ↔ z1 link of A2.

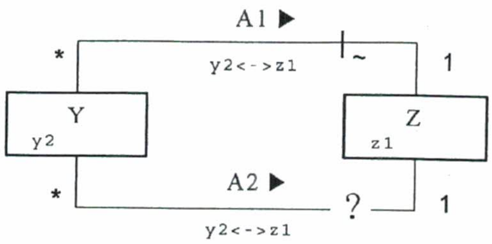

Fig. 11. Assocation cycle z1 ↔ y2, y2 ↔ z1.

Note that changing the multiplicity for Z in the A2 association from **1** to **0..1**, would not change the above scenarios. Also, if the binding for Y in association A2 was |- instead of default, the delete of z1 would always be unsuccessful.

For Fig. 11, Case 2, assume the "?" is replaced by a default implicit destructibility binding.

- If A1 is processed first, trying to delete z1 again causes an implicit destruction of the y2 ↔ z1 link of A1 and an implicit delete on y2. This will again be successful and result in the implicit destruction of the y2 ↔ z1 link of A2. Now, when A2 is processed to see if any links require implicit destruction, again none is found. Thus, the deletion of z1 is successful.
- If A2 is processed first, trying to delete z1 causes an implicit destruction of the y2 ↔ z1 link of A2. This would seem to result in a multiplicity violation of the lower bound **1**. However, no action is taken on this violation at this time, instead another check on this constraint is deferred to the end of the complex object operation, i.e., the commit of its encompassing nested transaction. (The related object y2 is inserted into *t.LbChecks*.) Next, A1 is processed, which causes an implicit destruction of the y2 ↔ z1 link of A1 and an implicit delete on y2, which will be successful. At commit of the complex object operation, no constraint violation for A2 is found since y2 does not exist (i.e., since y2 is not in *t.LbChecks* - *t.Deletes*) and, thus, the deletion of z1 is successful.

Fig. 12 is an example of an association cycle involving links of the same association.

- If the y1 ↔ z1 link of A1 is processed first, trying to delete z1 causes an implicit destruction of this link and an implicit delete on y1, which will be successful and result in the implicit destruction of the y1 ↔ y2 link of A2. Now, when the y2 ↔ z1 link of A1 is processed, this link is implicitly destroyed, and an attempt is made to delete y2, which will succeed

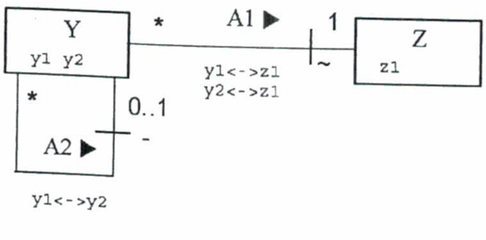

Fig. 12. Assocation cycle z1 ↔ y1, y1 ↔ y2, y2 ↔ z1.

since the y1 ↔ y2 link no longer exists. Thus, the deletion of z1 is successful.

- If the y2 ↔ z1 link of A1 is processed first, trying to delete z1 causes an implicit destruction of this link and an implicit delete on y2, which will be unsuccessful because the |- binding prevents the destruction of the y1 ↔ y2 link of A2. The complex object operation will be rolled back.

Again, note that, if the binding on the * end of association A2 was |-, the delete of z1 would always be unsuccessful.

A second problem with association cycles is evident from the above examples. They can cause the outcome of a complex object operation to be dependent on the order in which associations and links are processed. This can occur because cycles provide two alternate processing paths from one object to another and those paths can have different semantics. Fig. 11, Case 1, shows that outcomes can be dependent on the order in which different associations are processed, and Fig. 12 shows that outcomes can be dependent on the order in which the links of a single association are processed. When processing order is unspecified or indeterminate—as it is in relational database definitions and formal mathematical notations, both involving iterations over (unordered) sets—undesirable anomalies can occur when within an implementation and an ordering must be selected [20].

There are many ways to avoid this unpredictability. The following list borrows from [20].

1. Ideally, we could redesign the language or notation so that there is no loss in functionality and the processing order does not matter.
2. We could somehow allow the user to specify the processing order when it matters.
3. The system could try all possible processing orders at runtime and always fail if any of them fail (or always succeed if any succeed).
4. Cases where the processing order may matter could be detected at definition time and be disallowed.

The reader can probably discern the relative merits of each of these solutions. In the evolution of ORN, we have used solution 1 and currently employ 2 in a minor role.

Fortunately, only the |- (no implicit destruction) binding of ORN can cause processing order dependencies, and this is so only when it is given for just one end of an association involved in an association cycle. This is evident in the previous scenarios and is formally proven in the appendix . The |- binding is similar to the RESTRICT referential integrity rule in SQL [21], [22]. Unlike the RESTRICT, however, the |- binding can be protected from a "rear attack" by specifying this same binding for both ends of an association. In Figs. 11 and 12, when the |- binding is given for both ends of the A2 association, dependencies on the order of processing are eliminated, and the delete of z1 is always unsuccessful.

Use of the |- on only one end of an association, i.e., a *one-ended* |-, is often desirable and harmless, even in the presence of association cycles, which is why it is not simply disallowed. In Fig. 2, a one-ended |- is used for two associations, where no unpredictability results even though cycles are possible. For example, employee e1 supports child c1, who is also supported by e2, who is married to e1 (e1 ↔ c1, c1 ↔ e2, e2 ↔ e1). When a one-ended |- results in unpredictability, solution 4 above could be adopted to disallow it, but this was not done in OR+ since possible cycles are not inevitable and a warning can be issued.

Also, solution 2 can be employed when a one-ended |- results in processing order dependencies, which is hopefully rare. In OR+, a user can indirectly specify and predict the ordering in which associations and links are to be processed. Associations for an object are processed in the order in which their associated object-valued attributes are declared in the object's class (Fig. 3), and links for an association are processed in the order in which an iterator over a multivalued, object-valued attribute (or collection) returns references to the related objects. To control this ordering, the user must use an ordered collection, e.g., a List versus a Set, to implement the association.

This solution, however, is not highly desirable; hence, cases of processing order dependencies should be avoided. Sometimes they can be avoided by replacing a one-ended |- binding with a default implicit destructibility binding and **1** multiplicity. In some respects, this combination is similar to the NO ACTION referential integrity rule in SQL [21] and, as seen in the previous scenarios, avoids any order dependency problems.

5 CONCLUSION

ORN is a simple yet powerful notation for declaring association semantics at a very high level of abstraction, the entity-relationship, or object-association level. The use of this notation can enhance database development productivity and database integrity.

This paper has presented algorithms that can be used to implement ORN. We have given them at a level of abstraction that is independent of the type of database system, object or relational, and have successfully implemented them in OR+, an object DBMS prototype.

This paper has also explored the problems posed by association cycles. We have shown how circularity is avoided by the detection of such cycles in the given algorithms and that ORN semantics are predictable, and thus unambiguous, in their presence. That is, the outcomes of complex object operations are independent of the order in which association links are processed, except for one problematic specification. This is the one-ended |- binding given for an association that **may** have links that are part of an association cycle which **may** cause processing order dependencies. When such dependencies cannot be avoided, a user can control the processing order of links in the OR+ implementation of ORN, thus eliminating any ambiguity.

APPENDIX

Here, we state and prove the theorem that ORN semantics are unambiguous assuming a restriction on the |- binding. The theorem is stated and proven only in terms of object deletion; however, the corresponding theorems and proofs for association destruction and change are similar.

Theorem. *If no one-ended |- bindings are given for associations having links that are part of an association cycle, then the outcome of deleting an object under ORN is independent of the order in which links are processed.*

Proof. If the object being deleted and all objects linked to it directly or indirectly are not part of any association cycle, then there is only one processing path to any related object or link and thus only one possible outcome.

If, however, the object being deleted or any object linked to it directly or indirectly is part of one or more association cycles, then there can be multiple processing paths to related objects and links. We must show that the result of a complex object delete will be unaffected by the order in which links are processed. We do this by showing that the result of executing the *DeleteObject* algorithm, invoked in a nested transaction t to delete an object x in database d (as described in Section 3), is unaffected by the order in which the links of x or any related object are processed. This result, denoted by R, is defined by whether or not exit was with exception, and if not, the set of links that have been destroyed, denoted by *t.Destroys*; the set of objects that have been deleted, *t.Deletes*; and the set of objects remaining that must have lower bound multiplicities checked at commit, *t.LbChecks - t.Deletes*.

Let o be x or any object that is related to x directly or indirectly. Assume that prior to the invocation of $DeleteObject(x, t, d)$, o has n links to related objects, $o \leftrightarrow o_1, o \leftrightarrow o_2, \ldots, o \leftrightarrow o_n$. The links may involve one or more association types, the n related objects may not all be unique and may in fact be o, and o may be part of one or more association cycles.

If o is being explicitly deleted ($o = x$), then prior to $DeleteObject(o, t, d)$, none of o's links have been implicitly destroyed. If, however, o is being implicitly deleted, then one of its links, the *entry link*, has already been implicitly destroyed—e.g., in Fig. 11, Case 1, when A1 is processed first, the y2 ↔ z1 link of A1 for object y2. Furthermore, if o is part of one or more association cycles, then before a link can be processed by $DeleteObject(o, t, d)$, it may have become a *return link*. A return link is one that has already been implicitly destroyed as the result of the attempted deletion of a related object in an association cycle—e.g., in Fig. 11, Case 1, when A1 is processed first, the y2 ↔ z1 link of A2 for object z1. Without association cycles, the entry link does not change and there are no return links. With association cycles, whether or not a specific link is a entry or return link and, thus, has already been destroyed before its normal processing in $DeleteObject(o, t, d)$ is processing order dependent. Therefore, to show processing order independence, we must show that R will be unaffected if any link, $o \leftrightarrow o_k$, $1 \leq k \leq n$, has already been destroyed before it can be processed by $DeleteObject(o, t, d)$. We consider below each component of R and in *DeleteObject*, all possible cases of *impB*, the implicit destructibility binding for the o object class in the association of which $o \leftrightarrow o_k$ is a link.

Exit with exception. The only situation in which *DeleteObject* exits with an exception is when a |- is detected. This occurs in case "|-" when detected in the immediate invocation and case "|~" when detected in a recursive invocation. First, assume the |- binding. If $o \leftrightarrow o_k$ has already been destroyed, then an exception has already occurred because of the |- binding for the o_k object class. Here, we have applied the theorem's hypothesis. Now, assume the |~ binding, an *lB* violation, and that invocation of *DeleteObject* on o_k results in an exception. If $o \leftrightarrow o_k$ has already been destroyed, then a *DeleteObject* has already been invoked on o_k, resulting in the same exception. Note that, if o_k has been implicity deleted as a result of a ' binding, is uncommitted, and will subsequently be undone (i.e., we are in an nT transaction that will be aborted), then the results of $DeleteObject(o, t, d)$ will also be undone.

t.Destroys. In all cases, $o \leftrightarrow o_k$ is destroyed. Thus, *t.Destroys* is unaffected if $o \leftrightarrow o_k$ has already been destroyed.

t.Deletes. Cases "|~" and "'" may implicitly delete o_k, thus adding it to *t.Deletes*; however, if $o \leftrightarrow o_k$ has already been destroyed, then a *DeleteObject* has already been invoked on object o_k, and o_k is already in *t.Deletes*.

t.LbChecks - t.Deletes. Cases "none" and "'" add o_k to *t.LbChecks* if *lB* is violated. If, however, $o \leftrightarrow o_k$ has already been destroyed, then o_k is already in *t.Deletes* and, therefore, it is immaterial that o_k is not added to *t.LbChecks* since it will not be in *t.LbChecks - t.Deletes*. We have already shown that the *t.Deletes* component of R is unaffected by the order in which links are processed.

Since we have shown that all components of R, the result of executing $DeleteObject(x, t, d)$, are unaffected by the order in which links are processed, the theorem is proven. □

ACKNOWLEDGMENTS

This work was partially supported by the US National Science Foundation under grant CDA-9313299 and cooperative agreement HRD-9707076. Portions of Section 2 are reprinted from [12] with permission from the publisher, © 1996 ACM 0-89791-826-6. All rights reserved.

REFERENCES

[1] OMG Unified Modeling Language Specification, Version 1.3, Object Management Group, www.omg.org, Mar. 2000.

[2] W. Kim, "Object-Oriented Databases: Definition and Research Directions," *IEEE Trans. Knowledge and Data Eng.*, vol. 2, no. 3, pp. 327-341, Sept. 1990.

[3] Database Language SQL, Am. Nat'l Standards Inst., Inc., New York, www.ansi.org, 1999.

[4] B.K. Ehlmann and M.A. Stewart, "Incorporating Object Relationship Notation (ORN) into SQL," *Proc. 35th ACM Southeast Conf.*, pp. 282-289, Apr. 1997.

[5] B.K. Ehlmann and G.A. Riccardi, "A Comparison of ORN to Other Declarative Schemes for Specifying Association Semantics," *Information and Software Technology*, vol. 38, no. 7, pp. 455-465, July 1996.

[6] A. Albano, G. Ghelli, and B. Orsini, "A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language," *Proc. 17th Int'l Very Large Data Bases Conf.*, pp. 565-575, 1991.

[7] V.M. Markowitz, "Referential Integrity Revisited: An Object-Oriented Perspective," *Proc. 16th Int'l Very Large Data Bases Conf.*, pp. 578-589, 1990.

[8] J. Rumbaugh, "Controlling Propagation of Operations Using Attributes on Relations," *Proc. ACM OOPSLA*, pp. 285-296, Sept. 1988.

etected in a
ling. If o ↔
eption has
for the o_k
theorem's
B violation,
sults in an
ved, then a
esulting in
n implicity
nitted, and
in an nT
results of

yed. Thus,
eady been

y delete o_k,
has already
eady been
eletes.
" add o_k to
↔ o_k has
in t.Deletes
t added to
Deletes. We
ent of R is
ocessed.
s of R, the
affected by
theorem is
□

National
and co-
Section 2
publisher,

Version 1.3,
0.
nd Research
ol. 2, no. 3,

., Inc., New

ect Relation-
theast Conf.,

RN to Other
Semantics,"
pp. 455-465,

Mechanism
ogramming
pp. 565-575,

An Object-
Bases Conf.,

tions Using
35-296, Sept.

[9] C.J. Date, "Referential Integrity," *Proc. Seventh Int'l Very Large Data Bases Conf.*, pp. 2-12, 1981.

[10] B.K. Ehlmann and G.A. Riccardi, "An Integrated and Enhanced Methodology for Modeling and Implementing Object Associations," *J. Object-Oriented Programming*, vol. 10, no. 2, pp. 47-55, May 1997.

[11] P.P. Chen, "The Entity-Association Model: Towards a Unified View of Data," *ACM Trans. Database Systems*, vol. 1, no. 1, pp. 1-36, 1976.

[12] S.K. Hardeman (B.K. Ehlmann, advisor), "Association Behavior in Object Databases: Subtleties and Inconsistencies," *Proc. 34th ACM Southeast Conf.*, pp. 224-229, 1996.

[13] B.K. Ehlmann and G.A. Riccardi, "Object Relater *Plus*: A Practical Tool for Developing Enhanced Object Databases," *Proc. 13th Int'l Conf. Data Eng.*, pp. 412-421, Apr. 1997.

[14] R.G.G. Cattel, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade, *The Object Database Standard: ODMG 2.0.* San Mateo, Cailf.: Morgan Kaufmann, 1997.

[15] ObjectStore C++, Release 5, eXcelon Corp., Burlington, Mass., www.exceloncorp.com, 2000.

[16] B.K. Ehlmann, "A Data Modeling Tool Where Associations Come Alive," *Proc. 21st IASTED Int'l Conf. Modelling, Identification, and Control*, pp. 66-72, 2002.

[17] B.K. Ehlmann ORN Simulator, www.siue.edu/~behlman, 2001.

[18] B.K. Ehlmann, N. Rishe, and J. Shi, "The Formal Specification of ORN Semantics," *Information and Software Technology*, vol. 42, no. 3, pp. 159-170, Elsevier Science, 2000.

[19] B.K. Ehlmann and X. Yu, "Extending UML Class Diagrams to Capture Additional Semantics," *Proc. 20th IASTED Int'l Conf. Applied Informatics*, pp. 395-401, 2002.

[20] C.J. Date, *Relational Database Writings 1985-1989.* Reading, Mass.: Addison-Wesley, pp. 119-125, 143-147, 1990.

[21] C.J. Date and H. Darwen, *A Guide to the SQL Standard*, third ed. Reading, Mass.: Addison-Wesley, pp. 399-401, 1994.

[22] B.M. Horowitz, "A Run-Time Execution Model for Referential Integrity Maintenance," *Proc. Eighth Int'l Data Eng. Conf.*, pp. 548-556, 1992.

Bryon K. Ehlmann received the BS and MS degrees in computer science from the University of Missouri at Rolla in 1970 and 1971, respectively, and the PhD degree in computer science from Florida State University in 1992. He was formerly a full professor in the Department of Computer Information Sciences at Florida A&M University and a visiting researcher at the Supercomputer Computations Research Institute at Florida State University. Currently, he is an associate professor of computer science at Southern Illinois University at Edwardsville. Before earning the PhD degree, he worked for a number of years for Burroughs/Unisys Corp., where he did research and development on semantic data models, DBMSs, and query languages, and was also an assistant professor of computer science at Chapman University in Orange, California. His current research interests include data modeling, object-oriented databases, and software engineering. He a member of the ACM and the IEEE Computer Society.

Gregory A. Riccardi received the PhD degree in computer science from the State University at New York in Buffalo. He is a full professor of computer science at Florida State University and a recipient of the University Teaching Award. He is also currently a visiting research scientist at the UK National e-Science Centre in Edinburgh, Scotland, a faculty associate of the School of Computational Science and Information Technology at Florida State University, and a member of the Hall B Collaboration at the Thomas Jefferson National Accelerator Facility. He has published numerous research papers and three books. His *Database Management with Web Site Development Applications* was published in August 2002 by Addison Wesley. He is a member of IEEE and the IEEE Computer Society.

Naphtali D. Rishe received the PhD degree from Tel Aviv University in 1984. From 1984 to 1987, he was an assistant professor at the University of California, Santa Barbara. He is currently a full professor in the School of Computer Science at Florida International University (FIU) and director of the High Performance Database Research Center (HPDRC). He has published two books on database design, more than 120 papers, and holds three patents. He has been awarded over $17M in research grants by government agencies and industry. His current research focuses on efficiency and flexibility of database systems, distributed DBMSs, high-performance systems, database design tools, and Internet access to databases. He is a member of ACM and IEEE Computer Society.

Jinyu Shi received the BS degree in information engineering from Xidian University, Xian, China, in 1991. In 1999, he received the MS degree in software engineering science from Florida A&M University. He is currently a senior programmer/analyst in the Department of Direct Banking at Bank of America, Jacksonville, Florida. Before coming to the US, he worked for a number of years as a network engineer for several computer companies in P.R. China. His current research interests include component software, object-oriented analysis and design, software engineering, middleware technology, and Enterprise Application Integration (EAI).